Immersive iPhone Applications

MacTech Magazine
February • 2009

MACTECH®

The Journal of Macintosh Technology

DEBUGGING ON OS X

TORQUE GAME ENGINE

MORE LOCAL MCX

COCOA WITH PYTHON

CREATION

The Measure of the Mac

MACTECH.COM

$8.95 US, $12.95 Canada

02

0 32128 74887 8

ISSN 1067-8360 Printed in U.S.A.

Network, Server and Appliance Monitoring.

For Mac OS X

Xserve RAID

Xserve (Intel & G5)

Airport

Lithium Network Monitoring Platform

Lithium can now monitor your Xserve, Xserve RAID, Qlogic switches, Airports, Mac OS X Server... and everything else in your network.

www.lithiumcorp.com

lithium Corp.

LIBERATE YOUR LAPTOP

A SINGLE SOURCE SOLUTION PROVIDER
FOR ALL YOUR TECHNICAL REQUIREMENTS

PARTS + UPGRADES + SERVICE

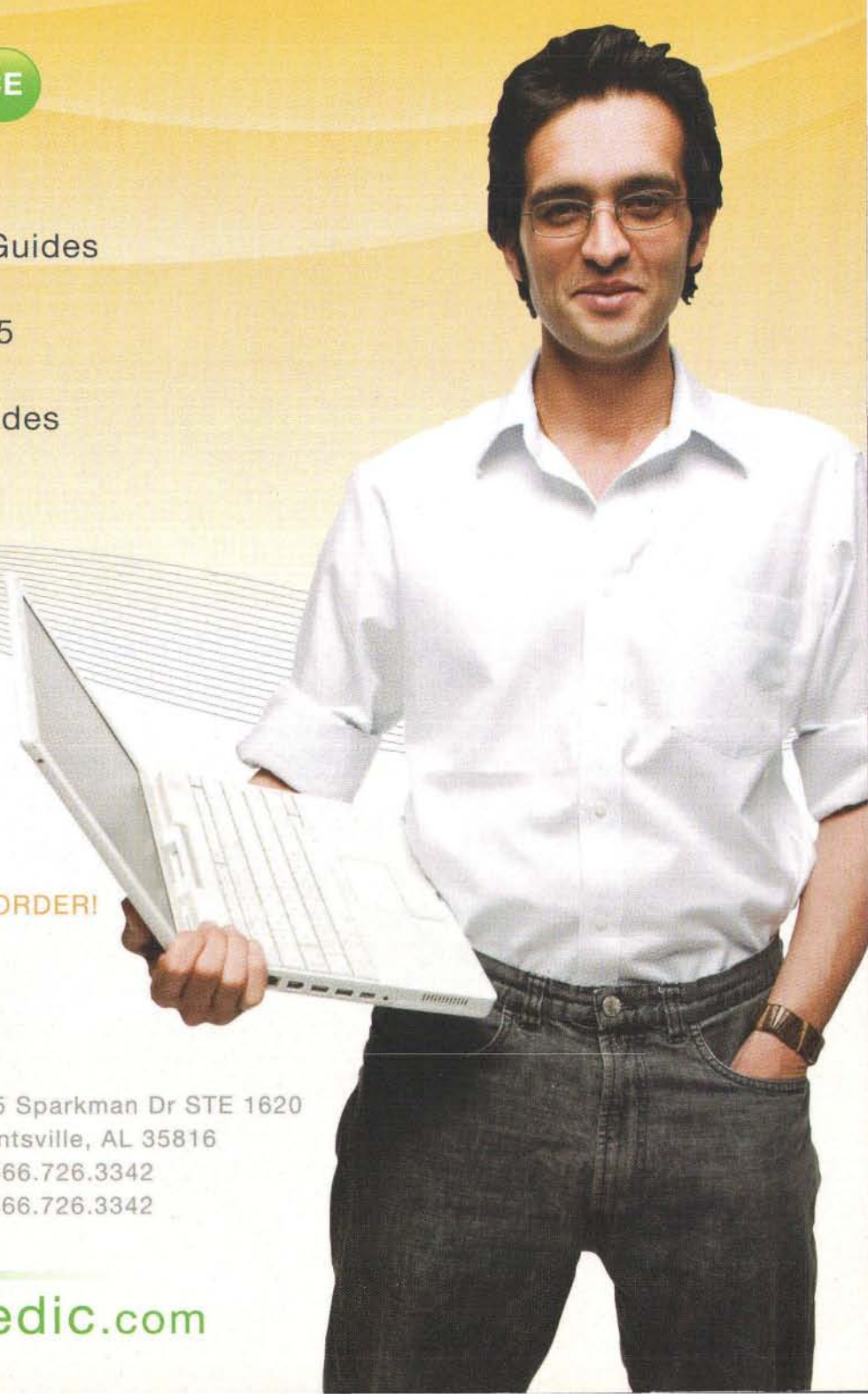
INDIVIDUALS

- Free Online Do-It-Yourself Guides
- Send-In Repair Service
- Power Adapters from $24.95
- Batteries from $94.95
- Replacement Parts & Upgrades
- Diagnostics
- Hardware Installation

DEALERS

- Reseller Accounts
- Outsource Repairs, Increase Profits
- Blind Drop Shipments

USE COUPON CODE MACTECH5
TO RECEIVE **5% OFF** YOUR FIRST ORDER!



PowerbookMedic.com

Your Mac — Our Patient

555 Sparkman Dr STE 1620
Huntsville, AL 35816
T 866.726.3342
F 866.726.3342

www.PowerbookMedic.com

The bulk-free scanner to de-bulk your life.

Meet the world's smallest color duplex scanner designed just for the Mac.

With a footprint that's half the size of a sheet of paper, the Fujitsu ScanSnap S300M will cut your workload—and your desktop clutter—down to size. The ScanSnap S300M has an automatic document feeder that holds up to 10 pages and scans both sides of everything from business cards to legal-size documents at a rate of up to 8 pages per minute. Now you can convert stacks of paperwork from receipts, recipes, even a budding artist's masterpiece, into PDFs with the touch of a button. And the ScanSnap S300M is Leopard compatible, with a choice of AC adapter or portable USB power so you can stay organized no matter where life takes you. Tell us how you'll ScanSnap. You just might win a $100 American Express gift check. Visit us at **http://us.fujitsu.com/scanners/tech**

ScanSnap S510M

ScanSnap S300M • AC adapter and USB power settings • Scans paper sizes from 2 x 2 inches up to Legal length • Automatic document feeder holds up to 10 pages • One button scanning to PDF and JPEG • Scans at speeds up to 8 pages per minute • Automatic color, de-skew, orientation and blank page removal • Bundled with Cardiris™ business card software • Includes Leopard compatibility • Duplex color up to 600dpi • Less than 3.1 lbs. • Only $295

Mac Universal

ScanSnap

Advance EXCHANGE

FUJITSU

THE POSSIBILITIES ARE INFINITE

CDW MacConnection MacMall Insight


ETYMOTIC
hf5
GEN
The next generation of in-ear earphones
You know your music.
Etymotic knows your ears.
amazon.com
visit: amazon.com/EtymoticResearch
©2008. hf5 is a trademark of Etymotic Research, Inc. The earphones are covered by one or more of the following U.S. patents: #4,677,679; #4,763,751; #5,887,070; #5,933,507; and other patents pending.

Amp Up Your Stuff!™

Crank up your savings!
Max out your Mac & iPod!

6 Months Same as Cash!
Offer valid for purchases over $500. Call for details.

Up to $200 Cash Back!*
On Apple computers. After mail-in rebate.

FREE Parallels Desktop!*
After mail-in rebate with purchase of an Apple computer.

FREE Printer!*
After mail-in rebate with purchase of an Apple® computer.

SAVE $100!
15" MacBook™ Pro 2.4GHz
with 250GB Hard Drive and 2GB SDRAM
FREE Parallels Desktop!* FREE Printer!*

~~$1994~~ - $100 mail-in rebate* = **$1894!*** #7684020
After mail-in rebate. See below for details.

SAVE $594!
13" MacBook™ Air 1.6GHz
80GB Hard Drive
FREE Parallels Desktop!*
FREE Epson Printer!*
original price $1794
~~$1299.00~~ - $150 = **$1199.99!*** #7373085
After mail-in rebate. See below.

SAVE $644!
15" MacBook Pro 2.4GHz
9600M SuperDrive
FREE Parallels Desktop!*
FREE Epson Printer!*
original price $1994
~~$1499~~ - $150 = **$1349.99!*** #7691239
After mail-in rebate. See below.

SAVE $50!
13" MacBook™ 2GHz
SuperDrive™
FREE Parallels Desktop!*
FREE Epson Printer!*
~~$1294~~ - $50 = **$1244!*** #7684018
After mail-in rebate. See below.

SAVE $35!
Microsoft Office 2008
for the Mac Student
and Teacher Edition
list price ~~$149.95~~
now **$114.99!** #7352258

SAVE $76!
1TB LaCie Big Disk
Extreme+ USB 2.0,
FireWire 800 and
FireWire 400 Hard Drive
was ~~$259.95~~
now **$183.94!** #7274662

Authorized Reseller

MacMall®
Your #1 Apple Superstore!

Source code: MACTECH

Call 1-877-233-2838 or visit macmall.com

*CASH BACK-Purchase select computer models shown on these pages and receive up to $150 cash back via MacMall mail-in rebate. Ends 2/28/09. • FREE PARALLELS DESKTOP OFFER-Get Parallels Desktop 4.0 for Mac free after $20 mfr. and $60 MacMall mail-in rebates with purchase of any new Apple computer. Price before rebates is $80. Ends 2/28/09. • FREE PRINTER OFFER-Get a printer FREE after $99.99 combined mail-in rebates with Apple CPU purchase. Price before rebates is $99.99. Ends 2/28/09. • ALL OFFERS VALID WHILE SUPPLIES LAST. Download rebate coupons at www.macmall.com/rebates. For rebate terms and conditions, please visit our Web site and enter the applicable part number.

TABLE OF CONTENTS

From the Editor

The issue of MacTech you're now reading is a long time in coming. Too long, perhaps. At MacTech, we simply strive to bring you the best tech knowledge to help you achieve your goals. This month, we're focusing on creation.

Creativity has been a hallmark of the Macintosh platform during its life. That implies, of course, that something gets created. This may be a new device, an application or a process. But *someone* had an idea and saw it through to reality. Someone may have even created an article that inspired others to use that knowledge and create further. It starts somewhere, though, and the more realms of knowledge that the creator is tapped into, the easier it will be to create and further mold that creation.

As a reader of MacTech, you can certainly create *something*: a shell script; an Automator action; an Objective-C function; an MCX record that changes that behavior of target machines; a one-line AppleScript that displays information in the GUI; *something*, if you realize it or not! It's useful to use existing skills as building blocks for larger challenges.

Also, you should recognize this creation in everything you do with technology: someone designed and created iMovie. Someone designed and created Photoshop. Someone designed and created the specification and functions that make up the MP3 codec. Someone created the computer (thank you, Alan Turing). There's little more important, in the technological scheme of things, than this creation.

In each month's MacTech Spotlight, we feature people that have a penchant for this creation. We ask about their inspiration, and what keeps them coming back for more. We want nothing more than a MacTech reader to be the next person featured in our Spotlight. So, what are we giving you this month to achieve this?

First, start with the Creation article. It delves into this concept a bit further, and features some known voices weighing in on the subject. From there, each article this month aids you in creating and working with your creations.

One special article this month is "Torque it to the Macs," which is an introduction to the Torque game engine. First, we want to illustrate that creating a product on the Mac doesn't always involve firing up Xcode. Also, realize that you're rarely bound to using a product in the way the creator intended. While GarageGames expects people to use Torque for game development, you are free to use it to create interactive training materials, simulations or 'edutainment,' or whatever you choose.

All of our regular columns are present, plus Macworld 2009 coverage, iPhone development tutorials and more. We do have two absences, though: Noah Gift's Samba/LDAP series, and Norman Palardy's REALbasic tutorials; both will resume next issue.

Apple has succeeded by never having a "comfort zone" causing them to constantly innovate and create. And, as a Mac user, Apple offers tools that allow *you* to create: Pages, Garage Band, iMovie, Keynote and more. At MacTech Magazine, we're going to be pushing past our comfort zone and offering deeper and broader support for all Macintosh-related technologies. *You* can influence this directly: send us your ideas and what you'd like to see covered: letters@mactech.com is always listening!

Ed Marczak,
Executive Editor

Communicate With Us

Department E-Mails

Orders, Circulation, & Customer Service
cust_service@mactech.com

Press Releases
press_releases@mactech.com

Ad Sales
adsales@mactech.com

Editorial
editorial@mactech.com
(Authors only, no pr)

Accounting
accounting@mactech.com

Marketing
marketing@mactech.com

General
info@mactech.com

Web Site
http://www.mactech.com

In this electronic age, the art of communication has become both easier and more complicated. Is it any surprise that we prefer **e-mail?**

If you have any questions, feel free to call us at 805/494-9797 or fax us at 805/494-9798.

If you would like a subscription or need customer service, feel free to contact MacTech Magazine Customer Service at 877-MACTECH

We love to hear from you! Please feel free to contact us with any suggestions or questions at any time.

Write to letters@mactech.com or editorial@mactech.com as appropriate.

MACTECH®

The Journal of Macintosh Technology

A publication of XPLAIN CORPORATION

Canada Post: Publications Mail Agreement #41513541
Canada Returns to be sent to: Bleuchip International, P.O. Box 25542, London, ON N6C 6B2

MacTech Magazine (ISSN: 1067-8360 / USPS: 010-227) is published monthly by Xplain Corporation, 5776-D Lindero Canyon #189, Westlake Village, CA 91362. Voice: 805/494-9797, FAX: 805/494-9798. Domestic subscription rates are $47.00 per year. Canadian subscriptions are $59.00 per year. All other international subscriptions are $97.00 per year. Please remit in U.S. funds only. Periodical postage is paid at Thousand Oaks, CA and at additional mailing office.

POSTMASTER: Send address changes to **MacTech Magazine**, P.O. Box 5200, Westlake Village, CA 91359-5200.

CREATION

Go make something – it'll make you a better tech, no matter your specialty

by Edward Marczak

Introduction

What *is* MacTech Magazine? There are certainly deeper questions in the universe, but this one is certainly pertinent to me and the readers of this magazine (you!). MacTech is certainly unique in that it straddles a fair range of disciplines; from interested techie, to system administrator to developer: we don't discriminate. Why is this question so pressing? Perhaps it is now more than ever.

Today, MacTech online and in print is read by 150,000 readers in 175 countries. Unfortunately, we rarely hear from the bulk of you. When we do, though, it's often about wanting a certain topic covered, or, to tell us that the content is starting to lean in a direction that feels skewed to "not me." Let's overcome that.

In the beginning...

A little background: I'm currently 38 years of age. So, I'm not the oldest technologist around, but in the computer field, I'm certainly aged. I'm fine with that, as I have experience and a perspective that people just getting into the field often do not. I also consider myself fortunate for being in the right place at the right time. I got to ride a very unique wave in watching computers develop. Yes, I've used punch cards and TTY based machines, modems with acoustic couplers and 300-baud "direct" modems. I've dialed into BBSes, surfed BitNet, telnetted into MUDs, retrieved files via gopher and shoved packets over protocols other than TCP/IP and mediums that were not twisted pair Ethernet. What good memories.

Attending university, I didn't really have much choice in a program to take, as computers were still relatively new and being fleshed out. If you were interested in computers, you took Computer Science, rife with math, assembly, Pascal and C. C++, Python, Ruby and Java either didn't exist or simply were not taught at the time, and System Administration was not a formalized discipline.

Here's where these stories come together: in my first job(s) out of school, IT (called "MIS," or, Management Information Systems, at the time), was still evolving. No matter your title, you were pretty much a techie. You wore many hats, from system and database administration to hardware guru to developer. I'll admit that, in the late 1980s and early 1990s, the scope of all those functions was certainly much smaller and more knowable. However, thanks to the evolution of Unix, System Administrators knew how to code; In C, typically. In the late 1990s, I worked with a gentleman that was the system administrator for a lab of SGI IRIX workstations used for graphics production. He also wrote custom shaders and effects for the custom 3D software they were using. Show of hands: how many of us do that today? OK, perhaps we don't need to go quite that far.

After reading over that, however, you should ask yourself what practical ways there are to increase your value. If you're a developer, learn a new API, better understand the low-level hardware that your code runs on or, better yet, learn how to package your final product in a way that makes it easy for end users and system administrators to install. If you're a system administrator, setting up a server, adding users and setting permissions are all fine activities. But when you have 100 (or more) users, have you thought about automating this process? What is your answer to "can we drop file X in each user's home directory?" If it's "really? That will take so much time!" you can be doing better.

Go create. Choose the itch, then scratch it.

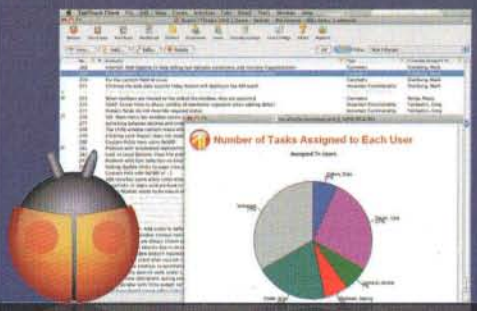
And really, *that* is why MacTech is here. We like talking about all of these issues. We enjoy the practical and the esoteric (which, when it comes to OS X, admittedly are sometimes the same). Sys Admins: you're *better* when you can script (in any language). You're better when you understand the file system structure, and you're better when you understand a problem due to a log entry you found that makes sense because you've been studying Cocoa. Developers: you're

Seapine Software

Seapine ALM solutions for serious Mac OS X development

TestTrack Pro

Issue & Defect Management



TestTrack TCM

Test Case Planning & Tracking

Surround SCM

Software Configuration Management

Stay on track with Seapine's Mac OS X-native development tools

Designed for the most demanding software development environments, Seapine's Mac OS X-native application lifecycle management (ALM) solutions are scalable, feature-rich, team-based tools that can be used separately for superior issue tracking, test case management, and software configuration management—or seamlessly integrated for more efficient control of your software development process.

TestTrack Pro

Issue & Defect Management

- Track defects, change requests, feature requests, and other project-related issues.
- Tailor workflows, including events, states, and transitions, to your development process.
- Stay informed and on track with flexible reports, email notifications, and escalation rules.
- Create and link defects with failed test runs in TestTrack TCM for better traceability.
- Link defects and change requests to source code changes in Surround SCM and other SCM tools.

TestTrack TCM

Test Case Planning & Tracking

- Manage thousands of test cases, select sets of tests to run against builds, and process the pass/fail results using your development workflow.
- Ensure all steps are executed, and in the same order, for more consistent testing.
- Know instantly which test cases have been executed, what your coverage is, and how much testing remains.
- Track test case execution times to more accurately estimate the time required to test applications.

Surround SCM

Software Configuration Management

- Control who changes source files, and track what has changed and when.
- Leverage file-level workflow to track and manage the state of individual files.
- Facilitate parallel development with advanced virtual branching, private branches, and workflows.
- Notify team members of new files, assignments, and changes by email.
- Quickly access the latest files with shadow folders, hyperlinks, and Finder integration.

Download evaluation copies of TestTrack Pro, TestTrack TCM, and Surround SCM at www.seapine.com/mactech

better when you understand how all of the parts outside of your application interact with the systems of end users (can you say, "FileVault" and "network home directories?"). You're better when you sit with the person responsible for installing your software on hundreds of machines and then watching it run.

MacTech is here to make you *better*. If you are interested in a particular topic, but haven't seen it, *let us know*! But very often, a solution comes from what we now consider to be a separate discipline. To check my beliefs, I started this very conversation with some of the top developers and system administrators today.

Nigel Kersten, Mac Operations Tech lead at Google summed up this philosophy more succinctly by saying, "sysadmins should learn more from software engineering principles. No script is too small to benefit from functional/unit tests, peer review, version control, release cycles and bug reporting systems. This all holds true for pretty much everything sysadmins do." So, when MacTech publishes an article on version control systems (git, subversion, and so on), it's not just for developers. Really!

Reinforcing this point was Dave Schroeder, Senior Systems Engineer at the University of Wisconsin-Madison. "There's a lot of crossover between many tools and techniques that have value in both developer and sys admin circles. Apple has taken this same approach with WWDC, combining developer and sys admin content into one conference. Some argue these should fundamentally be separated - but that can be costly, especially for a platform that is still in its relative infancy in the enterprise realm. There's more benefit at present in keeping such content together because of the overlap that often exists: a lot of highly technical topics on the Mac OS X platform have common roots."

When asking Marcus Zarra, Owner of Zarra Studios and blogger for Cocoa is my Girlfriend, he stated, "[f]rom my point of view, Mac Tech needs to cover the entire gambit of the Macintosh now. It is getting bigger and I am starting to see it in companies more often. Getting into companies means that we need sys admin content to help teach the new sys admins. The Macintosh is no longer just about the artist or the independent developer." Yes! It's a great time to be a part of the Mac scene. Market share for Apple products increases daily. OS X shows up in more and more places—expected and unexpected. There are more switchers—business and personal—every day. MacTech is interested in providing technologists the knowledge to contribute meaningfully to this new ecosystem.

Thomas A. Limoncelli, author of "Time Management for System Administrators," (O'Reilly and Associates) says, "The #1 thing people can learn is to automate processes. Whether that means learning AppleScript, Perl, Python, or Puppet. People often think of automation as a way to save time, but automation also prevents errors and/or adds consistency, which is more valuable."

Finally, Mark Dalrymple, Software Engineer and author, generalizes what to learn this way: "[Learn] everything :-) If you're not learning *something* new, you're stagnating, which is no fun. Eventually you get left behind, and everyone else gets to play with the new toys."

Don't write off your trusted MacBook®.

Introducing Axiotron® Modservice™ — transform your *existing* Apple® MacBook into an Axiotron Modbook®.

It works like this: Sign up for Modservice.* Choose your upgrade options and the warranty you want. Turn in your computer to one of our Axiotron Authorized Service Providers for conversion. Then kick back and enjoy your new Modbook. Draw, sketch and write by putting a pen to the screen of the best tablet computer with industry-leading Wacom® Penabled® technology and Mac OS® X. Portable and versatile, the Modbook empowers your creativity and imagination.

With Axiotron Modservice, your Apple MacBook computer gets revitalized, renewed and revolutionized — into a whole new product, the award-winning Axiotron Modbook. Visit www.axiotron.com/modservice for details.

the one and only AXIOTRON modbook

*Some service limitations and restrictions apply.

Get Involved

Getting involved can mean several things. First, *go create.* Choose the itch, then scratch it. Second, *have the conversation.* Which conversation? Any. Blog about your experiences, get involved on mailing lists, in forums and meet your peers at events large and small. There is also one more way: send us at MacTech an e-mail, and let us know what you'd like to see. letters@mactech.com is read by all editors; we're listening. MacTech has been in print in one form or another since the first Macintosh shipped (1984). We realize that times and trends change, and we're all in the midst of a sea change, right now. There's some great content already lined up for upcoming issues, but let us know how all of this is affecting *you*, and it will be reflected in our content. We're looking to fuel your knowledge, which creates more value for you.

About The Author

Ed Marczak knew even from the punch card and TTY days that technology was in his future. He finds all technology interesting, but chooses OS X when possible. When not computing, he spends time with his wife and two daughters.

If you have a smartphone, we can sync it.

The Missing Sync product family connects the coolest devices with Mac OS X. Reliably synchronize your address book, calendar, notes, music, pictures and more between your smartphone and your Mac.*

visit www.markspace.com/reliable

*Available for Blackberry, Windows Mobile, Palm OS, Apple iPhone, and Symbian OS devices.
Smartphone features vary from model to model, so synchronization features and capabilities will vary from product to product.
The Missing Sync is a registered trademark of Mark/Space, Inc.

mark/space

by Edward Marczak

Learning Python on the Mac: Functions

Modularizing and simplifying your code

Introduction

We've been learning Python on the Mac and have so far covered the basics. We need to introduce a little more foundation this month. *Functions*—in any language—allow the author to create reusable blocks of code. This is important from a few perspectives: code reuse, the ability to refactor easily and debugging. Functions lead to building libraries, both of which are critical concepts to becoming an effective Python programmer and part of an essential foundation for creating an OS X utility or application.

Jumping In

Let's get started. Like variables, functions don't need to be defined before use. Need a function? Just define it. Similarly, functions may be defined anywhere in the source file, and in any order. Since a function defines a code block, the rules of indentation apply: choose spaces or tabs (trying to be consistent with the style you use in the rest of your code) and indent the entire function. Here's an example:

```
#!/usr/bin/env python

def SayHello():
  print "Hello"

SayHello()
```

This short program simply prints "Hello." Not too exciting, and it really could have been accomplished in one line. But it does help illustrate the basics: A function is created by using the `def` keyword, supplying a function name—unique to this source file—followed by parenthesis and terminated with a colon character. The lines following need to be consistently indented; the first line that lowers the indentation level ends the code block that comprises the function. The previous example is really as simple as it can get. Let's look at something a little more useful: a function definition that prints the square of a number passed into it.

```
def Square(number):
  print number,"squared is",number ** 2
```

Once defined, this function can be called as many times as you have need, individually, or in a loop:

```
for i in range(1,6):
  Square(i)
```

This will produce:

```
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
```

However, there's a problem with this function. What if we only want to compute the value of a square and keep it for later, rather than print it out? That's where the `return` keyword comes in. `return` sends a value back to the caller. So, our `Square()` function could be rewritten like this:

```
def Square(number):
  return number ** 2
```

and would need to be called like this:

```
x = Square(44)
```

or, used in-line like this:

```
print "The square of 78 is %d." % (Square(78))
```

How, you may ask, is this any better than the first version? The answer is two-fold: first, you'll typically use functions to build up more complex sections of code. The times that you have a one-line or very simple function usually involve creating a function for readability purposes. Second, by returning a value, you can store it for later, rather than display it or otherwise use it immediately.

Libraries

Let's build up a small set of useful functions, and learn some new Python skills along the way. We'll improve and expand on these functions as time goes on. Also, for now, some

NEW LOOK. NEW PRICE. NEW YEAR.

NOW LAUNCHED - WELCOME ABOARD!

www.esellerate.net

of this is a little more advanced than we've covered, so, just trust me for now, and this will all be covered in future columns. Here's the beginning of the code, along with the first function, and yes, it's one that you just need to trust me on for now:

Listing 1a – Wrapper for subprocess

```
#!/usr/bin/env python

import subprocess

def RunProc(command,env_vars=None):
  """Wrap subprocess into something reasonable.

  Args:
    command: List containing command and arguments
    env_vars: Shell environment variables that should be
present for execution
  Returns:
    stdout: output of the command
  """
  proc = subprocess.Popen(command, stdout=subprocess.PIPE,
                          stderr=subprocess.PIPE,
                          env=env_vars)
  (stdout, stderr) = proc.communicate()
  return stdout
```

Note that I'm also shooting for good habits, here: commented code, readable variable names and consistent, clean code style (space after comma, spaces around equal signs, and so on). The above code wraps the subprocess library calls to make it easier to run a child process. It's less than perfect right now, but we'll correct that in due time (not this column, however). Let's get on to a useful OS X-related function.

Listint 1b – GetLocalUsers function.

```
def GetLocalUsers():
  """Fills a dictionary with all system users
     Args:
       None
     Returns:
       Dictionary, filled with uid/username pairs
  """
  command = ['/usr/bin/dscl','.','list','/Users']
  output = RunProc(command)
  userList = output.split('\n')
  userDict = {}
  for userName in userList:
    if userName is not '':
      command = ['dscl','.','read','/Users/'+userName,'uid']
      output = RunProc(command)
      uidLocation = output.split(':')
      uid = uidLocation[2].strip()
      userDict[uid] = userName
  return userDict
```

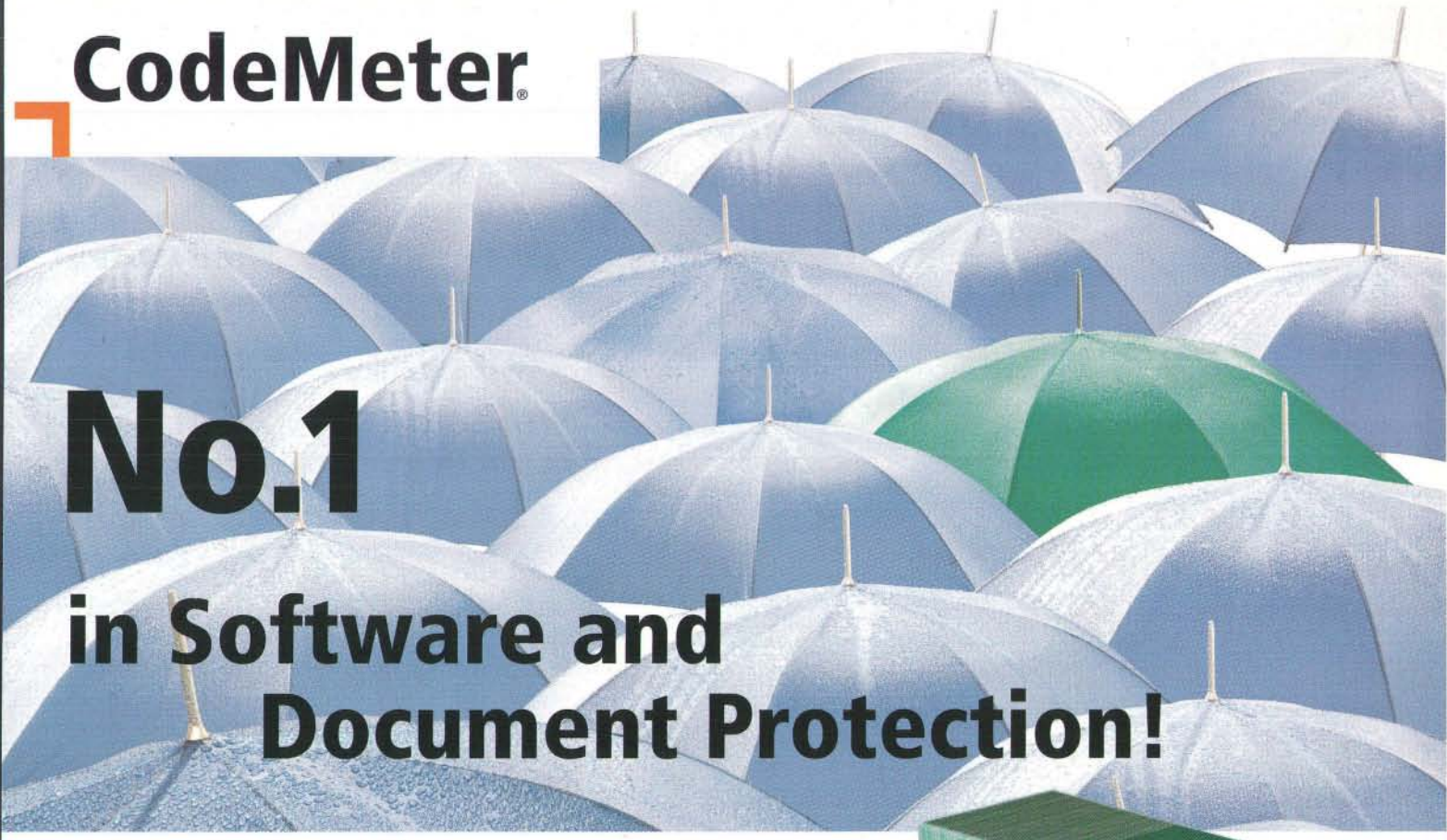
The GetSystemUsers() function will fill a dictionary with uid (the key) and the short name that it is assigned to. It takes no parameters, and can be called like this:

```
userDict = GetLocalUsers()
```

To extract information, simply request the value using the uid as the key:

```
print "User ID 74 is %s." % (userDict[74])
```

CodeMeter®



No.1

in Software and Document Protection!

Highest Security

- Vendor selectable secret and private key.
- Strong encryption algorithms with AES 128-bit and ECC 224-bit.
- Best-in-class tools for automatic protection (envelope, without source modification) for Win32, Win64, .NET, Java and MacOS X Universal (PPC, Intel).

Best Flexibility

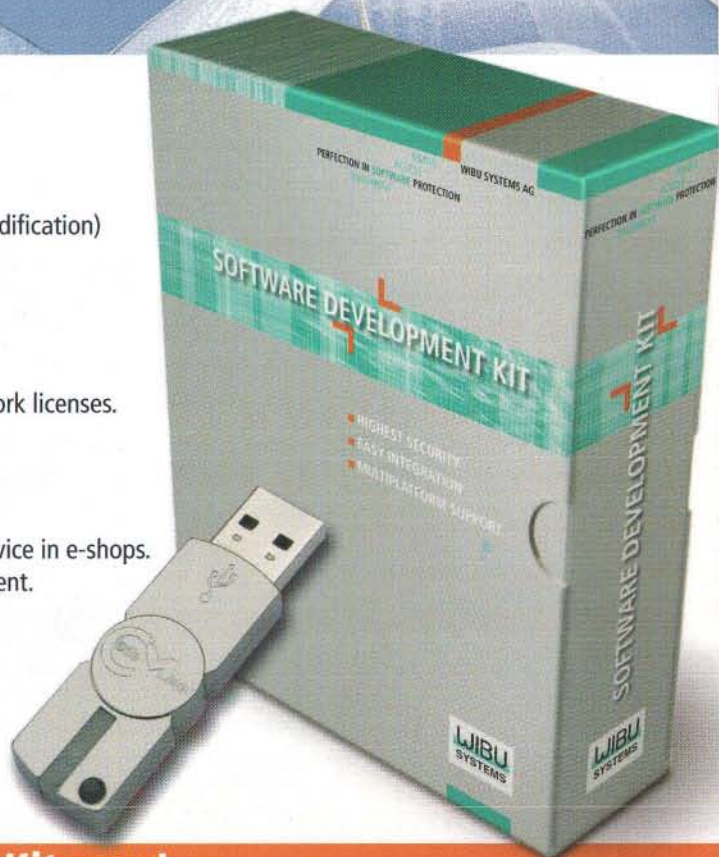
- More than 1000 independent licenses can be protected by one CM-Stick.
- One versatile hardware key for all license models including floating network licenses.
- Multi platform support including Windows, MacOS X and Linux.

New Distribution Channels

- License transfer by SOAP based CM-Talk or file based Field Activation Service in e-shops.
- Multiple-purpose, including protecting low cost software and digital content.

Unique End User Advantages

- First and smallest dongle with up to 2 Gbyte flash drive.
- No drivers necessary – can be used without administrator rights.
- CM Password Manager, secure virtual drive and secure login.



Order your Free Software Development Kit now!
Phone 1-800-6-GO-WIBU | order@wibu.us

2007 SIIA //CODiE// FINALIST
2005 SIIA //CODiE// FINALIST
2006 SIIA //CODiE// FINALIST

Mac

Universal

WIBU-SYSTEMS submitted the CodeMeter Password Manager and the CodeMeter SDK for the **Apple Design Awards 2007.**

WIBU SYSTEMS

WIBU®-SYSTEMS USA Inc.
110 W Dayton Street
Edmonds, WA 98020, USA
www.wibu.com
info@wibu.us
Tel: 1.800.6.GO.WIBU
1.425.775.6900
Fax: 1.206.237.2644

(It's _mysql, if you're curious and not typing in code while reading this).

There are some new concepts in Listing 1b. The `split` string function (line 10 of listing 1b), creates a list, each element created by the boundary of the character argument. For example, the string 'This is a string' when split using a space (`string.split(' ')`) becomes the following list:

```
['This', 'is', 'a', 'string']
```

Any character can be used to provide the boundary marker.

This function is nice, but could certainly be improved. What if we only wanted standard users, and not all of the system users? That's where function *arguments* come in. Like passing arguments into a command line application, functions can accept arguments, too. Alter our function to do so:

```
def GetLocalUsers(incSystemUsers):
```

This will allow us to pass a value into the function. Great, but we need to do something with it. Let's expect this to be a Boolean True or False: if True, include the system accounts, if False, do not. (I'm classifying accounts based on uid—only system accounts have a uid of less than 500). Update the code portion of the function to utilize this new variable (the first ten lines are the same as before—I just wanted to give some context):

```
command = ['/usr/bin/dscl','.','list','/Users']
  output = RunProc(command)
  userList = output.split('\n')
  userDict = {}
  for userName in userList:
    if userName is not '':
      command = ['dscl','.','read','/Users/'+userName,'uid']
      output = RunProc(command)
      uidLocation = output.split(':')
      uid = uidLocation[2].strip()
      if incSystemUsers:
        userDict[int(uid)] = userName
      else:
        if int(uid) > 499:
          userDict[int(uid)] = userName
  return userDict
```

We use a simple if/else statement to determine if the user should be included in the dictionary that is returned. We should update our comments about this in the function, too.

Now, we have to include a value when we call the function. Like this:

```
userDict = GetLocalUsers(False)
```

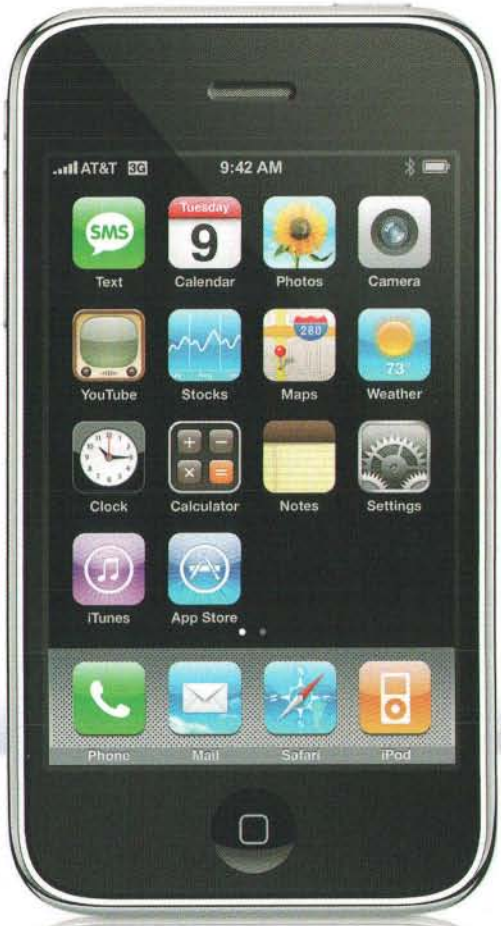
This is an improvement to this function. Frankly, though, more often than not, you probably do only want standard users.

Default Arguments

Python functions understand *default arguments*. Change the function definition to read:

```
def GetLocalUsers(incSystemUsers=False):
```

KERIO



iPhone & Kerio MailServer

Synced Wirelessly

Kerio MailServer for Mac OS X pushes email, contact and calendar updates to your new iPhone wherever you may be. With remote wipe, the mail administrator can delete sensitive information in the event your phone goes missing. Download the latest version for your risk-free 30-day trial or contact Kerio today.

1-888-77-KERIO | www.kerio.com

Now, if we don't tell GetLocalUsers how to behave, by not including an argument, it will assign False and continue on. So now, we can once again call the function with no arguments:

```
userDict = GetLocalUsers()
```

This is equivalent to calling `GetLocalUsers(False)`. We really only have to pass in a value of True if we wish to override the default behavior. It is possible to combine default and non-default arguments in a single function definition. For example:

```
def Mount(path, method="afp", mount_point="/Volumes",
shadow=False):
```

This fictional function, `Mount()`, has four parameters, only one which you are required to provide when called: `path`. The remainder use the defaults provided. Therefore, this function can be called in the following ways:

```
Mount("serv.example.com/share")
Mount("serv.example.com/share", "nfs")
Mount("serv.example.com/share","/tmp/mnt")
```

These examples use simple positional arguments: you need to know the position of the argument and pass in the values in the correct sequence. When creating a function definition, default arguments should be grouped at the end of the list. This allows them to be omitted.

Python also supports specifying the argument that is being passed in. This allows us to call the function like this:

```
Mount("serv.example.com/data.dmg","http",shadow=True)
```

Note that we're passing in the value for `shadow` in the *third* position, where we would typically specify `mount_point`. This is an excellent way to pass parameters into a function, as it makes it exceptionally obvious what is taking place. Which of the following code is easier to read?

```
ConvertFile('rawdata.txt','sales.csv','csv',True)
```

or:

```
ConvertFile(infile='rawdata.txt', outfile='sales.csv',
            format='csv',Totals=True)
```

Hmmmm? Using named arguments is a good habit to get into.

Variable Length Argument Lists

As a final discourse in functions, what if you don't know how many parameters a function may receive, or, has a number that may change from call to call? Python supports *variable length* argument lists. Two decorators designate how these values are received. If a single asterisk is used, the values are received into a tuple, receiving any excess positional parameters, defaulting to an empty tuple. If a double asterisk is used, it is initialized to a new dictionary receiving any excess keyword arguments, defaulting to a new empty dictionary. Let's see how this works. Here's a short, sample function that accepts any number of arguments:

```
def SomeFunction(*args):
  print args
```

It can be called like this

```
SomeFunction(24, 'blah', 'cold', 88, [22,34,66],'starch')
```

Note that arguments do not need to be of the same type. Pass in strings, integers, or even lists and dictionaries. The same goes for the double-asterisk decorator, with one exception: since you're creating a dictionary, you need to pass the key and the value.

Changing the function definition to:

```
def SomeFunction(**args):
```

allows the function to be called like this:

```
SomeFunction(number=24, some_string='blah',
             a_list = [22,34,66], string2='starch')
```

In our case, this will cause the function to output:

```
{'a_list': [22, 34, 66], 'some_string': 'blah', 'string2':
'starch', 'number': 24}
```

Finally, realize that you can mix variable arguments along with standard and named arguments. In the examples used above, since we only specified a variable argument, passing in an argument is completely optional.

In Conclusion

I was hoping to rip through functions and talk about modules in this column, but then I started writing, and realized the scope of the topic. I want to do justice to both functions and modules, as they're both critical foundational subjects. Next month, we'll build on the functions introduced this month and dig into modules.

Media of the month: I'm not going to call out any one particular 'thing,' but will give this directive: Find something new. If you've only been exposed to the Mac, go pick up a book (or find a web page) about Windows, FreeBSD or Linux. Compare to OS X. If you're a MySQL person, download and explore PostgreSQL, or learn the command-line interface to SQLite. You get the idea—stretch your boundaries.

About The Author

Ed Marczak is the Executive Editor of MacTech Magazine. He lives in New York with his wife, two daughters and various pets. He has been involved with technology since Atari sucked him in, and has followed Apple since the Apple I days. He spends his days on the Mac team at Google, and free time with his family and/or playing music. Ed is the author of the Apple Training Series book, "Advanced System Administration v10.5," and has written for MacTech since 2004.

Loco for Local MCX

Using Apple's client-management settings on the Local machine. Following up on MCX

By Greg Neagle, MacEnterprise.org

MacEnterprise.org
Mac OS X enterprise deployment project

Previously in MacTech...

In the November issue, we looked at getting started with MCX by using MCX records in the local directory service. We set some policies for the loginwindow using the Guest Computer object, verified they worked, and noted that the Accounts preference pane had certain controls grayed out to match our management settings.

Since local directory service entries in Leopard are simply plist files stored in `/var/db/dslocal/nodes/Default`, replicating MCX settings to multiple machines can be as simple as copying a few files.

Scalability and Modularity

If you are going to use the local directory service to store MCX data, you'll quickly discover that putting all your management settings into the Guest Computer object doesn't give you very much flexibility. If you want different groups of machines to have different management settings, you'll have to maintain multiple versions of `/var/db/dslocal/nodes/Default/computers/guest.plist`, and mixing and matching management settings becomes tedious, as you'd have to create a version of the guest.plist for every needed combination of management settings.

A new feature of MCX in Leopard can help: Computer Groups. Prior to Leopard, MCX supported "Computer Lists", which were, unsurprisingly, lists of computers. Management settings could be applied to Computer Lists, which would then apply those settings to all computers in the list. The downside is that a given computer could be a member of only a single computer list at a time.

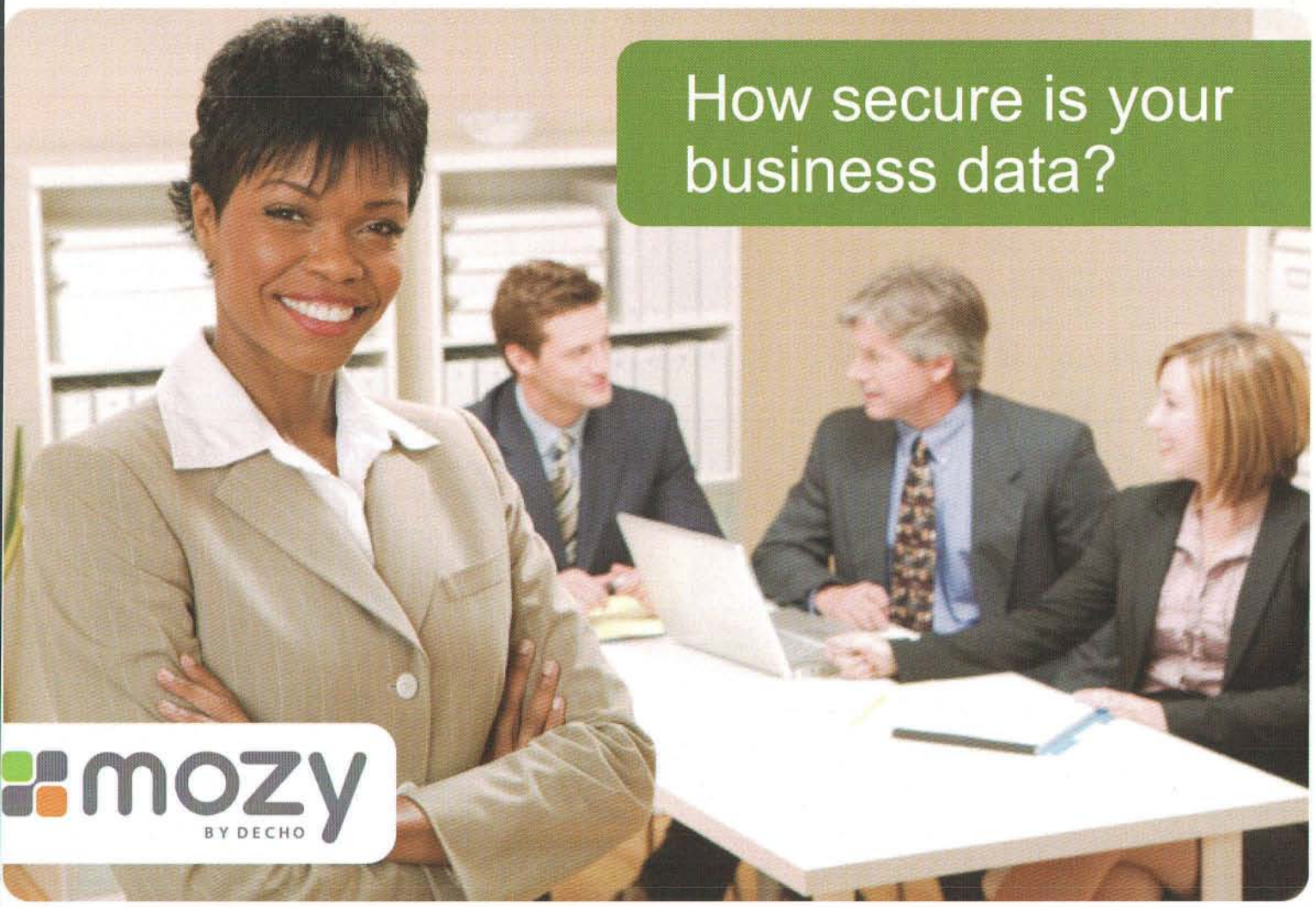
Leopard's new Computer Groups behave more like groups of users: a computer can be a member of multiple Computer Groups. This allows you to modularize your management settings. Create a Computer Group called "loginwindow" and apply all the loginwindow management to that Computer Group. Create another Computer Group called "screensaver" and apply your screensaver policies. If you want a machine to have both the "loginwindow" and "screensaver" policies, just add it to both Computer Groups. In this way, you can modularize your policy settings and mix and match them among machines. This approach even works if you are implementing MCX management in a traditional network directory environment.

Complications ensue

Perhaps now you're thinking, "Great! I'll create Computer Groups with my desired management settings, add the Guest Computer to each of those Computer Groups, and I'm all set!" Not so fast. There's a complication: even though Workgroup Manager allows you to add the Guest Computer account to a Computer Group, the MCX settings for the Computer Group are not applied to guest computers. So we'll need to use another way to take advantage of Computer Groups.

Use Workgroup Manager to connect to the local directory service as we did last time:

We will create a computer record that will represent the current local computer. Make sure you are working in the



Protect your business with MozyPro online backup

MozyPro is the simple and safe way to protect all the important files on your business computers. A copy of all your files is stored offsite in secure data centers, so you are always covered in the event of file corruption, accidental deletion, hardware failure or even natural disaster.

Why take the chance? Get started today and save! Visit us online at **www.mozy.com/mactech** or call your industry representative at **877.669.9776** and receive 10% off your initial purchase when you use the promo code **MACTECH10**.

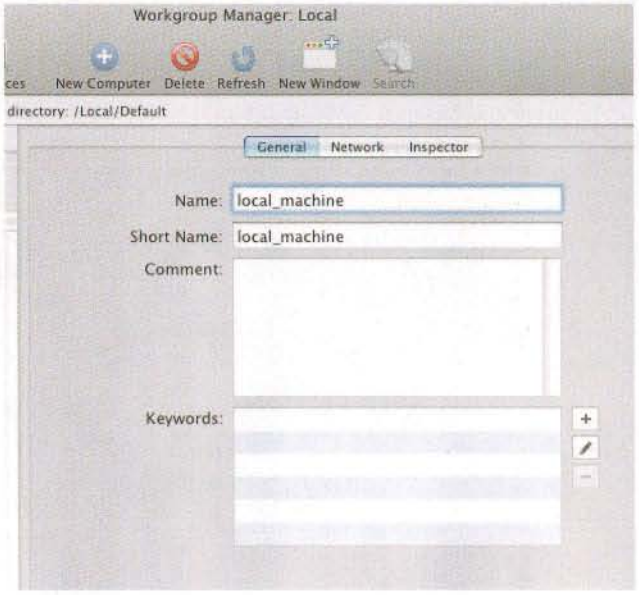
Simple. Secure. Affordable.

- Mac and Windows support
- Online account management
- Multiple restore options
- The highest levels of security
- Backup solutions starting at less than $5

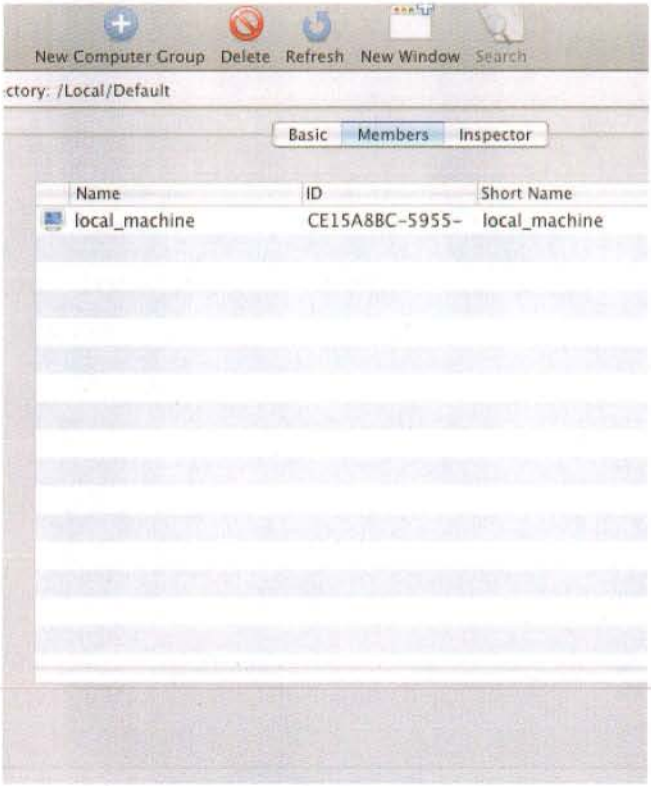
"MozyPro is the first online backup service I'm willing to offer my clients. No other company can offer as great of a service at such a great price."

- David O'Connell
OConnell I.T.

/Local/Default directory. Select the **Computer** accounts icon in the left pane, and click **New Computer**. Set the **Name** and **Short Name** to "local_machine", and click **Save**. You can leave all other fields empty.



You can now add this computer record to your Computer Groups:



We're not done yet – MCX still has no way to know that by "local_machine" you mean the current machine that this is running on. To do this, we have to add some data to the local_machine record.

MCX identifies machines by the MAC layer address of their en0 interface – this is usually the built-in Ethernet port, but in the case of a MacBook Air, this is the AirPort interface. Here's a way in the command-line to get this info:

```
root# ifconfig en0 | awk ' /ether/ {print $2}'
00:1b:63:93:8b:ac
```

So the MAC layer address for this machine is 00:1b:63:93:8b:ac. We can add that to the local_machine record like this:

```
root# dscl . -create /Computers/local_machine \
ENetAddress 00:1b:63:93:8b:ac
```

Checking our work:

```
root# dscl . read /Computers/local_machine
AppleMetaNodeLocation: /Local/Default
ENetAddress: 00:1b:63:93:8b:ac
GeneratedUID: 15BEE70A-A32D-4A33-B740-93CBE95F75A4
RecordName: local_machine
RecordType: dsRecTypeStandard:Computers
```

Note that the GeneratedUID will vary, as it is created when you create the computer record in Workgroup Manager.

To actually take advantage of this, we'll need to write a script that runs on each machine that modifies the local_machine record with the MAC layer address of the current machine. Here's a very simple version:

```
#!/bin/sh
MAC=`ifconfig en0 | awk ' /ether/ {print $2}'`
dscl . -create /Computers/local_machine ENetAddress $MAC
```

This script gets the current MAC layer address and updates the local_machine record accordingly. You'd want to implement this script as one that ran during startup. You could do this as part of a StartupItem, or use launchd.

Pulling it together

Now you have a computer object that corresponds to the local machine, and management settings attached to Computer Groups. To implement these management settings on another machine or machines, you need to copy to each machine:

- `/var/db/dslocal/nodes/Default/computers/local_machine.plist`
- The appropriate plists from `/var/db/dslocal/nodes/Default/Computer Groups`
- Your startup script that modifies the local_machine record.

Photofont

WebReady

A Revolution in Web Typography

Transform this...

Into this!

Photofont WebReady allows you to enhance your web pages with custom fonts of your choice in a search engine friendly, standards-compliant way. With **Photofont WebReady**, you can convert any photofont, OpenType font or TrueType font into an embedded web font. The web font is then rendered on your web page using Flash® technology, yet keeping all the advantages of standard hypertext. Your visitors see the page the way you want them to see it.

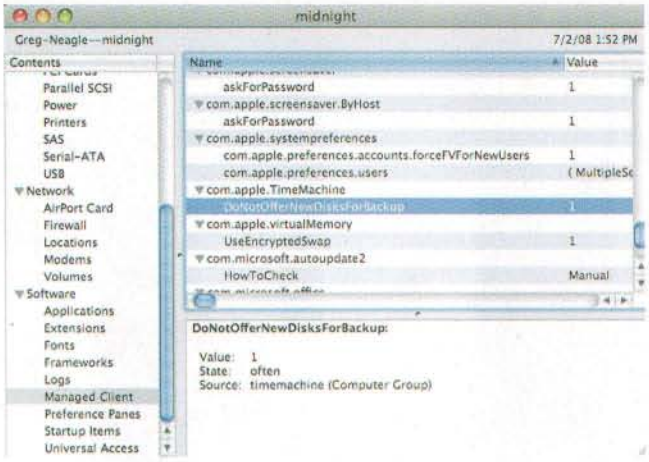
Learn more about **Photofont WebReady** and photofonts at **http://www.photofont.com/photofont/webready/**

1. The main title <H1> has been converted to a photofont to better match the desired look and feel of the site. With photofonts designers can take advantage of gradients, transparency, shadows and more!
2. Until now, web designers had to settle for Arial, Georgia or some other "safe" font to make sure the overall design of the page looked consistent. With **Photofont WebReady** designers can use the most suitable font for each project.

CRAFTED BY FontLAB

Testing and Troubleshooting

While testing your new MCX settings, you'll want to be aware of tools available to you so you can see which MCX settings have been applied to your machines. The easiest tool to use in Leopard is System Profiler. Under the Software section is a new Managed Client category. You can use this to see what preferences are being managed, and their source.



In the illustration, we can see that com.apple.TimeMachine's preference "DoNotOfferNewDisksForBackUp" is set to "1", that this is applied "often", and these settings come from membership in the "timemachine" Computer Group.

Another tool is available at the command line: mcxquery. Prior to 10.5.3, mcxquery did not display proper results when MCX records were in the local directory service, but it now behaves as expected.

```
mcxquery -user shortname
```

will show you the effective MCX settings for the user on the current machine.

Conclusion and additional thoughts

In this series of articles, we've looked at ways of using the local directory service to implement MCX settings. This would be useful for environments that don't have Open Directory, and cannot or will not do schema extension for Active Directory or third-party LDAP directory services. Mac administrators can also use these techniques as a proof-of-concept for MCX client management, using the local directory service implementation as a prototype for what benefits your organization would realize by extending the schema on its existing directory services.

There are certainly applications of this concept outside the management of enterprise workstations. Parents could use Workgroup Manager to manage their children's user account and capabilities – a sort of Parental Controls on steroids. In another application, the author used MCX records in the local directory service to lock down a Mac used as an iTunes jukebox. Since you are using local records, you can experiment with various settings and not worry that what you try will affect other machines on your network.

If nothing else, taking advantage of local MCX records should lower the entry bar to using Apple's client management system. If you haven't been using MCX in the past, there is no excuse not to at least try it now, and see how it can help you better manage Macs in your organization and give your users a consistent user experience.

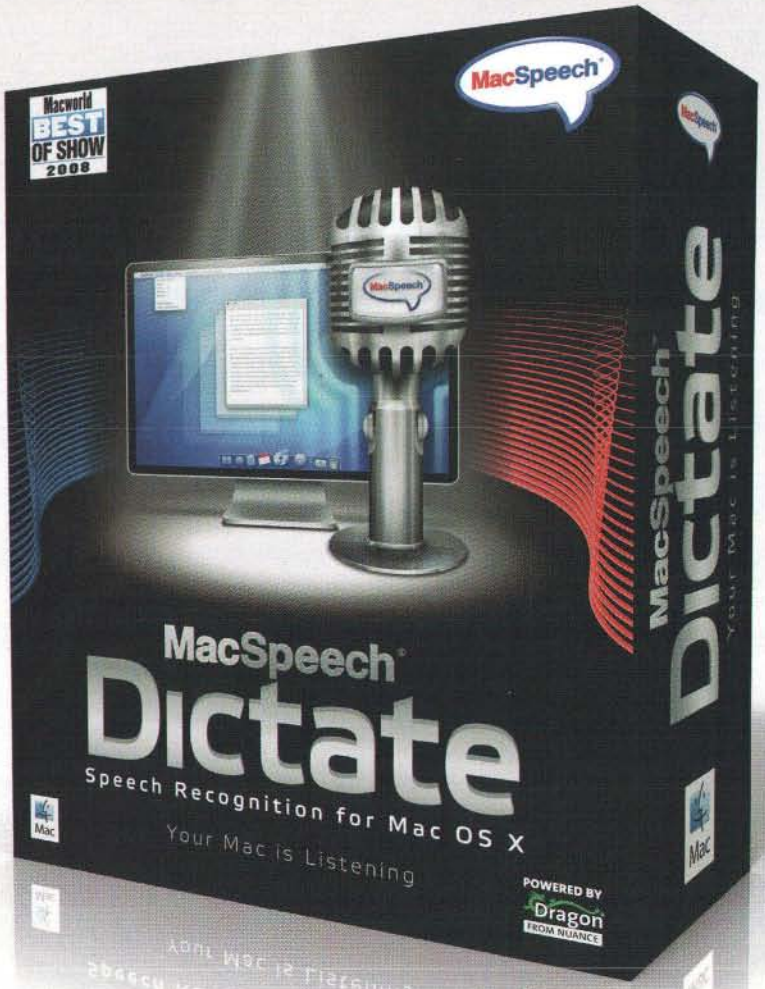
About The Author

Greg Neagle is a member of the steering committee of the Mac OS X Enterprise Project (macenterprise.org) and is a senior systems engineer at a large animation studio. Greg has been working with the Mac since 1984, and with OS X since its release. He can be reached at gregneagle@mac.com.

It's time to speak up.

Introducing MacSpeech Dictate 1.2 with Spelling and Phrase Training.

Speech recognition so good, about the only thing it can't do is speak for you.



MacSpeech®

www.macspeech.com

MACWORLD 2009: BEST OF SHOW

MacTech and Macsimum News team up for Best of Show awards

MacTech Magazine and Macsimum News teamed up for the first time to announce their Best of Show awards for the 2009 Macworld Conference & Expo. With so many products from over 450 exhibitors, editors from both publications combined forces to find interesting jewels on the show floor. In alphabetical order by product, here are the winners:

1Password, by **Agile Web Solutions**. 1Password is a password manager and 'data safe' that can store bank account information, personal details, login credentials and secure notes. Thanks to plugins for Safari, Firefox and other browsers, these values can be automatically filled in on web-based forms. An iPhone app is also available that can sync with the desktop version (http://agilewebsolutions.com/products/1Password).

BlueStor by **JMR.** BlueStor is a new line of Mac-compatible 16- and 8-bay RAID storage solutions. BlueStor uniquely uses a PCIe card and treats the RAID unit also as an external chassis that can in turn be used as a point of expansion for other PCIe cards, including further storage expansion. Powered by a Mac Pro using Final Cut Pro software, the BlueStor product line will run multi-stream real-time HD playback (http://jmr.com).

Dictate 1.2.1 by **MacSpeech** is the latest version of the speech recognition for the Mac. Dictate customers can dictate any specific word, no matter how obscure, by spelling it letter-by-letter with the Spelling mode. It also has a "move" command for easier verbal editing of a document, a Spelling Mode for letter-by-letter dictation of unusual words, and the Phrase Training capabilities that help the user increase accuracy on the fly. (http://www.macspeech.com)

FontXChange by **Morrison softDesign** is Mac OS X software that converts fonts to OpenType, PostScript Type 1 and TrueType for compatibility across both Macs and Windows systems. It supports batch processing for converting entire font libraries and has preferences to set font encodings. FontXChange has a font inspection window with preview and support for many font encodings. The latest version upgrades encodings to Unicode 5.1, updates parsing and rendering of various font tables, adds an option to generate .AFM files, adds font naming options and adds a feature to rebuild damaged fonts. (http://www.morrisonsoftdesign.com)

GraniteSTOR by **Small Tree** is a new family of shared storage solutions. First in Small Tree's GraniteSTOR product line is EasyAoE, an ATA over Ethernet protocol shared storage solution. It's designed specifically for audio/video editors and graphic artists looking for affordable means to manage shared storage solutions (http://www.small-tree.com).

iRecord by **Streamingnetworks** is a personal media transcoder that lets you record movies, songs, TV shows, video game clips and streaming audio directly to an iPod, PSP or USB storage device without a computer. It records using H.264 compression techniques while audio is encoded using AAC compression in real time (http://www.irecord.com).

NEC LCD3090WQXi. This is a 30-inch, 2600 x 1600 desktop LCD display for high-end graphics applications. It has a wide format design (16:10 aspect ratio) that provides roughly the same work area as two smaller sized displays. The NEC display offers a wide color gamut, an ambient sensor, MultiSync support and more (http://www.necdisplay.com).

ModBook Pro by **Axiotron** is an aftermarket modification of the unibody MacBook Pro, Axiotron's ModBook Pro offers a 15.4-inch pen-enabled display with a pixel resolution of 1440 x 900 WXGA, and features a surface layer of the company's etched, paper-emulating ForceGlass for drawing and writing comfort. It features Axiotron's Synergy Touch, a technology that employs touch functionality primarily in support of the pen input (http://www.axiotron.com).

Neat Receipts by **Neat Co.** Neat Receipts is a mobile scanner and digital filing system that enables you to scan receipts, business cards and documents so you can organize, store and secure all your important information. The patented technology identifies and extracts the important information and automatically organizes it for you. A Spotlight plugin allows you to search on all information without launching the application (http://neatco.com/products/neatreceipts-for-mac).

PhotoMotion by **GeeThree** is for Final Cut Pro and Final Cut Express lets you preview and import high-resolution images from existing libraries. You can preview images from Aperture, Lightroom, and iPhoto libraries and Pictures folders. Other folders can also be added. The plug-in appears as a standard Final Cut generator. There's no need to adjust keyframes when adding transitions or changing the duration of the animation (http://www.geethree.com).

LiveScribe Pulse by **LiveScribe.** Livescribe's Pulse Smartpen captures handwriting and simultaneously records audio and synchronizes it to the writing, so users never miss a word they hear, write or speak. Users can tap on their notes to replay what was recorded from the exact moment they were writing. With the Livescribe Desktop software, notes and audio recordings can then be transferred to a Mac, where they can be digitally stored, searched, or shared (http://www.livescribe.com).

TapIt4Me for iPhone by **Ettore Software**. This is a version of the company's text expander app (originally available only for the Mac) that runs on the iPhone and iPod Touch. The application expands keyboard mnemonics into full phrases and whole paragraphs, increasing your typing speed (http://ettoresoftware.com).

TypeTrax by **Insider Software** and **Moksa**. Type Trax is a new product for managing design projects created using Adobe Creative Suite. Developed jointly by Insider Software and Moksa, Inc., TypeTrax manages projects, and tracks, organizes and pre-flights project files, including fonts and digital assets. It adds the missing link to digital design projects. Fully integrated with Canto Cumulus, a leader in digital asset management, TypeTrax allows a user to simply drag a document into Canto Cumulus and TypeTrax automatically detects and displays the project's font resources in WYSIWYG mode (http://www.typetrax.com).

MORE PACKED!

The MacTech DVD - Volumes 1.01-24.12 is packed with more than ever before -- almost 3100 articles from more than 280 issues **(1984 - 2008)** written by over 850 experts, all 29 issues of Apple's develop, 21 issues of FrameWorks magazine, 100+ MB of source code, MacTech Viewer, working applications, full documentation, demos for techs, ***and more!*** The latest version includes all kinds of third party applications, videos, and demos.

Everything is displayed in the very fast, very searchable ***MacTech Viewer!*** An application that has been designed specifically with Techs in mind. Search quickly through almost 25 years of of great information provided by MacTech. Information to save you time, and make your life easier.

NEW VERSION!

Mac Universal

Requires Mac OS X v. 10.4.5 or later

Toll Free 877-MACTECH, Outside US/Canada: 805-494-9797 • http://www.mactech.com/dvd/

Python Cocoa - Delicious!

Creating Mac OS X applications using Python instead of Objective C

by Scott Corley

Python?

If you're like me, there was a time in your life that you enjoyed programming. And if you're like me, you sometimes wonder if it was all a dream.

It wasn't a dream. Programming is fun when you can get your ideas executing as fast as possible. If your programming time is spent making things work they way you imagine, and tweaking things to make them perfect, it is pure joy, like nothing else in the world. On the other hand, if you are spending your time writing function declarations in header files, then rewriting them again in definition files, and endlessly declaring properties and synthesized attributes and wrestling with the compiler and typing and typing and typing just to see 500 compiler errors and 200 linker errors... well, it's not fun anymore.

Enter Python. Python is a very powerful object oriented language like Objective C, with some important differences. The first thing you'll notice about programming in Python is that it is a hell of a lot of fun.

Getting Started

You probably already have everything you need to write complete Cocoa apps in Python instead of Objective C. If you have Leopard, you've also got an up-to-date Python 2.5 interpreter - it's a standard part of OS X. Hooray! And, if you have XCode 3.0 or later, it comes with Python Cocoa bindings in the form of something called PyObjC 2.0. Woot! And to top it all off, XCode 3.0 has support for Cocoa-Python Application project types. OMFG!

Add it all up and you're ready to go, without even swiping a mousepad finger. Of course, if you have an earlier version of XCode or OS X, you'll probably be missing some critical functionality that is relied upon herein, but odds are if you're reading this, you're up to date on all counts.

Let's get started. What we're going to do is take a pretty straightforward Apple Cocoa / XCode / Interface Builder tutorial—the good old Currency Converter—and go through it step by step. But instead of writing our classes in Objective C like Apple's tutorial suggests, we're going to do those parts in Python.

The cool thing is that everything in the tutorial works great, and as you'll see, once you've gone through this exercise with me once, you should be able to figure out how to do almost everything else in Python rather than Objective C.

Same Cocoa, Different Language

This article is going to make heavy use of an Apple Developer Connection: "Cocoa Application Tutorial: The Currency Converter Application". Go get it now! Browse to developer.apple.com and type "Currency Converter Application" into the search box. Click on the first result. You can download the whole thing in PDF format and print it out, or just follow along with the html version.

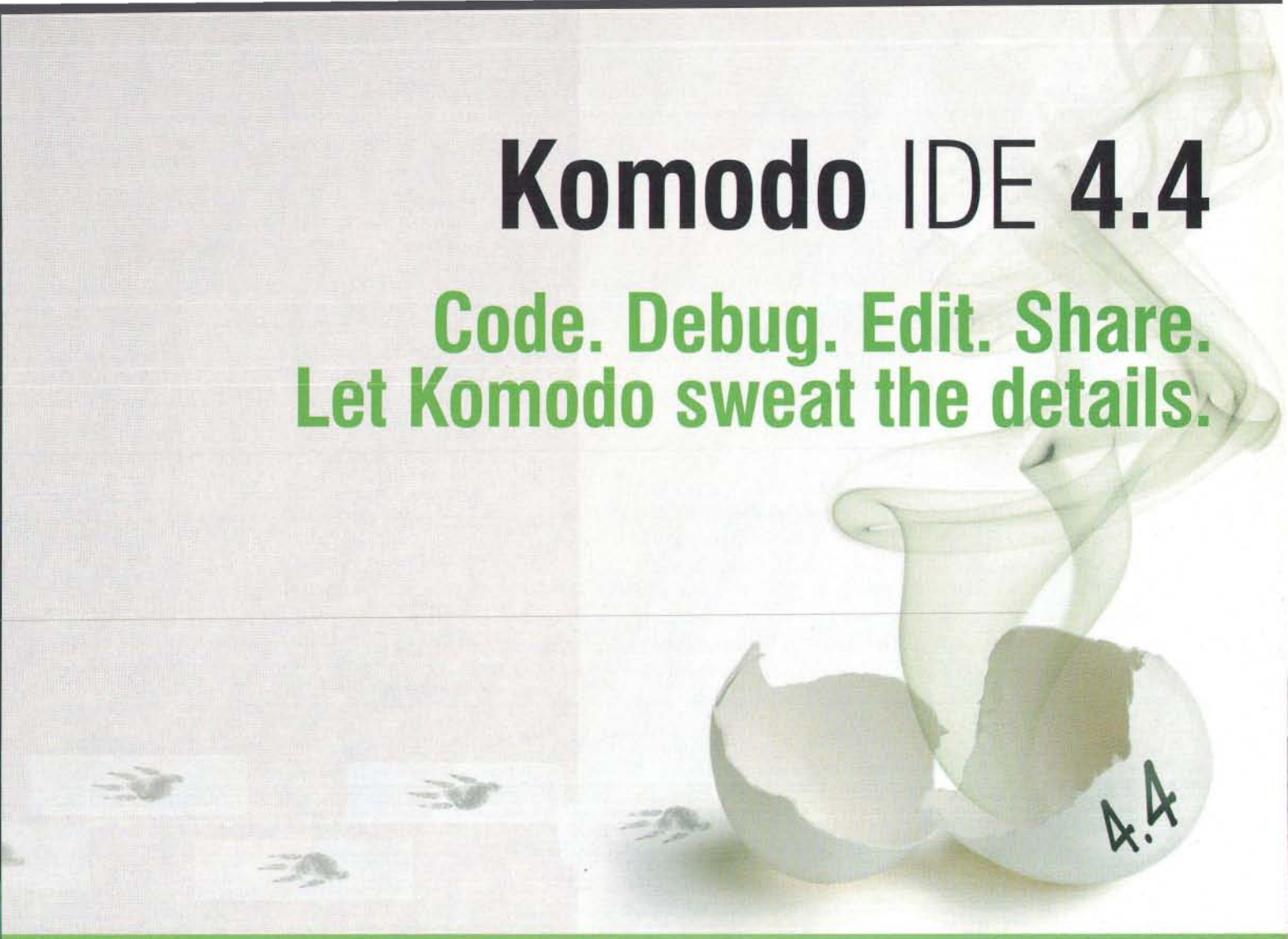
Once you have Apple's Currency Converter Application tutorial in front of you, skip the "Essence of Cocoa" section and go right to "Creating a Project in XCode". We're going to open XCode, and make a new project.

Open XCode, and once it's running, choose File > New Project. Apple's tutorial advises you to select "Cocoa Application" as the project type. Instead, select "Cocoa-Python" application. If you don't see Cocoa-Python application as an option, make sure "Mac OS X / Application" is highlighted in the left pane of the New Project window. You'll need to scroll down a bit to find Cocoa-Python Application. If it's still not there, make sure you have XCode 3.0 or later! It's in there, just dig around. There are actually four different types of Cocoa-Python project templates, we're after the simplest one that is just called "Cocoa-Python Application".

OK, now choose a location and a name for your project. I'm going to call mine Python Currency Converter. Once that is done, I am given a pretty normal looking XCode project. I see some familiar things, like a "main.m" file, a MainMenu.xib file, and an Info.plist. There are also some unfamiliar things, like a

Komodo IDE 4.4

Code. Debug. Edit. Share. Let Komodo sweat the details.



Code Intuitively

Create great applications using dynamic languages and open technologies with Komodo IDE's award-winning multi-language editor.

Code client-side and browser-side apps simply and naturally with advanced support for Perl, PHP, Python, Ruby, Tcl, HTML, CSS, JavaScript, XML, and more.

Work Smarter with Powerful Tools

Focus on what your code can do, not on fixing problems.

Find errors quickly with Komodo IDE's superior remote and multi-threaded debugging, solve tricky regular expressions with the Rx Toolkit, and get instant feedback from syntax checking and syntax coloring—Komodo IDE is jam-packed with intelligent tools to speed development.

Build and Organize with a Team

Get your project together efficiently with source code control integration, integrated project manager, and multi-user support.

Team development is more fluid with the security and flexibility of Subversion, CVS, and Perforce support. Share file templates, common configuration files, and toolboxes to unify and accelerate teamwork.

Develop Your Way

Automate, extend, customize: hack away! With Firefox-style extensibility, highly configurable run commands, and powerful macros, you have complete control of your IDE.

And you're covered wherever you want to work with Komodo IDE's flexible multi-platform licensing for Windows, Mac OS X, and Linux.

Download your free 21-day trial at www.activestate.com/MacTrial

ActiveState

"Python_Currency_ConverterAppDelegate.py" python file. Also, note that "main.m" is the only Objective C file in the whole project – and we intend to keep it that way. All of our other code will be in python files.

So, let's build and run this project to see Python in action. Hit Command-Enter, and the app should start up with a default menu bar, and an empty window named Window. Great! It's totally working!

Quit the Python Currency Converter test you just launched and go back to XCode. We're ready for the next step of Apple's Currency Converter tutorial.

Model in Python

In the next part, we'll be specifying the model class. Select File > New File in XCode. Instead of creating an Objective C file and a header file, choose "Pure Python" in the left pane of the New File dialog, then choose Python Module as the new file type, and click Next. Name the file Converter.py. Recall that Python does not have separate header files, so this one file is all you will need. Sweet! One of my favorite things about Python is ditching header files for good. Header files are only there to make the compiler's life easier, and they make our lives harder. Goodbye forever, header files!!!

Ahem. The file we just created, Converter.py, is going to contain a Python class that represents our data model. Our data model for this example consists of a currency amount, and a currency exchange rate.

We are going to create a class in Converter.py to represent our data model. This class will be very simple, and it will follow the one in Apple's example very closely. See Listing 1 for the code of Converter.py.

Listing 1: Converter.py

Currency Converter Model

This is our entire model class. Yes, the currency converter is simple, but note how little redundancy is needed in the structure of this code to express the underlying simplicity.

```
#
#  Converter.py
#  Python Currency Converter

class Converter:
    def __init__(self):
        self.sourceCurrencyAmount = 0.0
        self.rate = 1.0

    def convertCurrency(self):
        print "amount:", self.sourceCurrencyAmount
        print "rate:", self.rate
        return self.sourceCurrencyAmount * self.rate
```

Finding Errors

But wait! you say. I just tried to give myself an IndentationError, and instead I got a dreaded TERMINATING DUE TO UNCAUGHT EXCEPTION!! And the XCode debugger dropped me into some nasty assembly code! I THOUGHT YOU SAID THIS WAS BETTER!!!

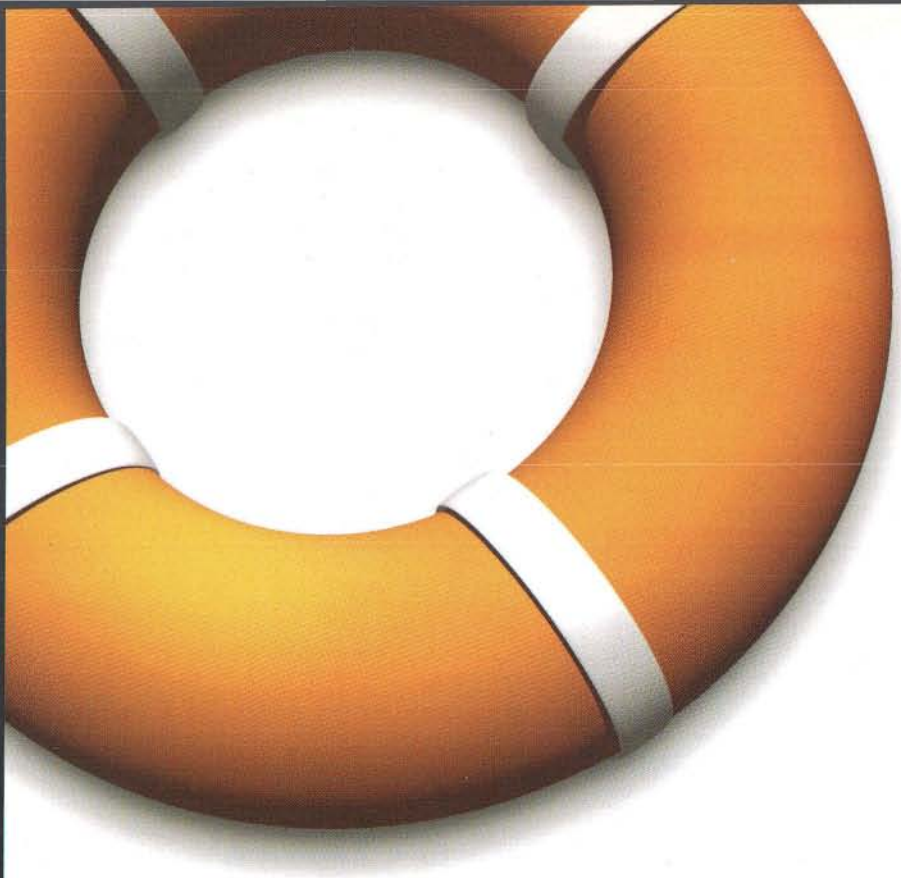
OK, ok, hang on. Two things. First, Python is giving you a very nice error message and returning an error code, but then our main.m file throws an NSInternalInconsistencyException exception in Objective C. Ultimately, while we will be doing 99.99% of our work in python, everything unwinds back to main.m, the little Objective C stub that gets our app going in the first place. When your Cocoa-Python app has a problem, the answer is usually in XCode's console.

To make your app to behave a bit more pythonic, have it exit with an error code instead of throwing an NSException. Edit the main.m file and look for the line that says "if (result != 0)", right after PyRun_SimpleFile. Just change the NSException call to an NSLog instead. A Python error will then cause the app to gracefully exit with the Python error code rather than boorishly shouting about an uncaught Objective C exception.

If you run your Cocoa-Python app from XCode and you encounter an error in Python, all of the wonderful information about what went wrong in Python will be displayed in the console. Select Run > Console to see the console output. You will need to scroll up a bit in the Console, past the gigantic GNU gdb copyright message, to see your Python traceback. Once you see the line "Traceback (most recent call last):" rejoice!!! To new python programmers, a traceback looks like a scary mess, saddening evidence that something has gone wrong. However, once you get used to reading tracebacks (they're pretty easy to understand), you will giggle with joy when you see a traceback. Sure, it is evidence that something has gone wrong, but it is damn good evidence that points right to the problem and says exactly what the problem is.

A crash in python can typically be fixed in seconds - after glancing at the traceback, you will slap your forehead, say "of course!" very loudly, edit the file to fix the problem, and be very confident that the fix is a true fix and not just a temporary workaround for a symptom of something worse. In my experience, other languages make it much tougher to find out what actually went wrong, when, where, and why. A crash in most languages is usually a sad event... "there goes my afternoon!" A traceback in python is a cause for celebration - "hooray, I have discovered something incorrect in my program, and it will soon be fixed forever!"

The fact that the incredibly useful traceback information is hidden away in the Console window takes some getting used to. XCode warriors are used to looking at call stacks in the debugger, with the occasional useful message in the Console. The XCode debugger has no Python support as of yet, so every time you flip the debugger open and there's nothing there, just remind yourself to go take a peek at the Console to look for a Python traceback.



Help Has Arrived.

Web Help Desk

- ✓ 100% Web-based Service Management Solution
- ✓ Incident & Problem Management
- ✓ Asset Life Cycle Management
- ✓ Knowledge-Base Management
- ✓ Apple Remote Desktop Integration
- ✓ LDAP / Active Directory Synchronization
- ✓ Approval & Change Workflow
- ✓ Customer Service Web Portal
- ✓ Reporting Features
- ✓ Licensed or Hosted SaaS Plans

Mac Universal Windows

MySQL ORACLE Microsoft SQL Server

www.webhelpdesk.com **Powerful Software for Support Management**

Since the converter class only represents the model for our application, and has no direct user interface interaction, it can be done entirely in Python. No PyObjC imports are needed. Even if the model for our application was more complex, it could still be represented with pure Python, relying on all of the power that the standard Python library provides. If we want to do Mac-specific tasks, we could import PyObjC into our model and call into Cocoa. In our case, though, it's not necessary, and Converter.py is fantastically simple.

The Converter class initializes the sourceCurrencyAmount and rate member variables, and it defines a single method, convertCurrency, to do the simple currency conversion.

That's all there is to Converter.py.

Before we move on, take a moment to enjoy the simplicity of Converter.py. We didn't have to declare any types, we didn't have to create a header file, we didn't have to declare our convertCurrency method in one place and define it in another. We didn't have to declare our member variables as properties, and we didn't have to tell the compiler to synthesize anything. We just defined what we wanted, in about as simple as a fashion as possible, and it's going to work great. That's pythonic.

User Interface

This is where things get cool. It's nice enough to be able to write python scripts on the Mac. But we really want a full-fledged Cocoa UI, and we want to be able to build it nicely in Interface Builder like civilized people.

XCode's Python support allows us to do exactly that. Moving on with Apple's Currency Converter tutorial, let's go to the section titled "Defining the View: Building the User Interface". Skip ahead to the section titled "Creating the Currency Converter Window".

Locate MainMenu.xib in your project. Double-click it to open it up in Interface Builder. Once Interface Builder loads it up, you will see your default menu bar and an empty window.

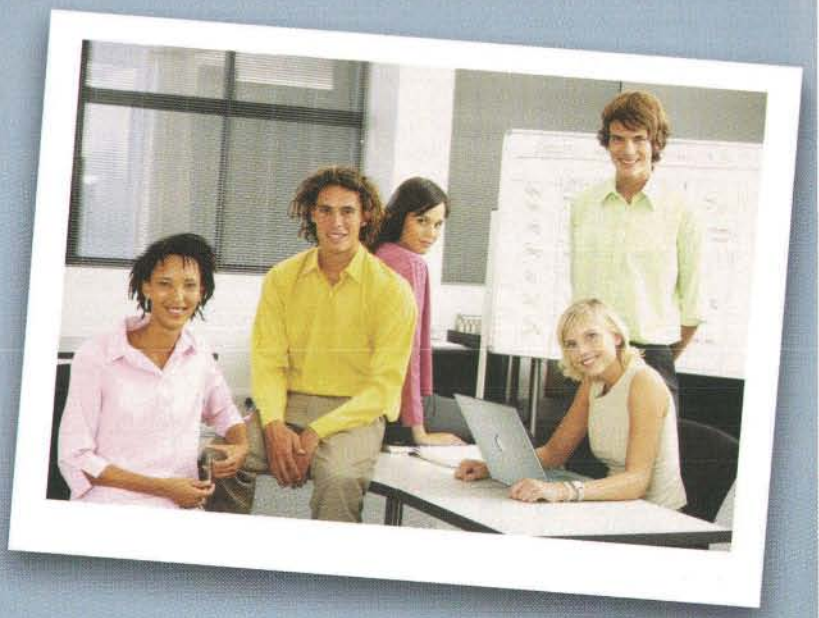
At this point, we're totally on-message with the Apple tutorial. We can lay out the interface exactly as described. If you prefer, you can download the layout from the MacTech source repository (ftp://ftp.mactech.com).

Once MainMenu.xib is set up with the three text fields, three labels, and the Convert button, save it and test it out in Interface Builder. Then run Python Currency Converter to see that the interface shows up in our Python app, and responds to key presses and mouse clicks. The interface is still not wired up to our Python code, but we're all ready to do that step.

Bridging the Model and View: The Controller

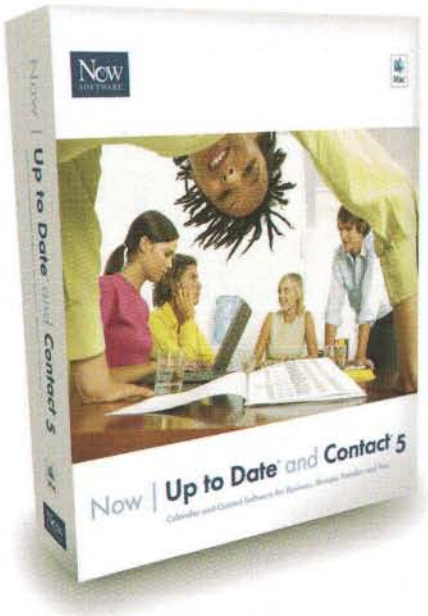
We're now on the part of Apple's tutorial that deals with Outlets, Targets and Actions. If you've built a Cocoa app, you are familiar with what these are and how to set them up in

Now **scheduling** and **contact management** for your entire organization.



Now | **Up to Date**® and **Contact**® **5**

Calendar and Contact Software for Business, Groups, Families and You.



Is this project on schedule? When are you available to meet about the systems upgrade? Where are all the field techs today? When was the last time anyone talked to our biggest customer?

Virtually all groups live (or die) by their abilities to meet deadlines and keep track of their customers, prospects, and vendors. Few small companies or even departments of big companies have the tools they need.

Now Up-to-Date & Contact might just be the calendar and contact software for you. It's time-tested and used by more Mac-based companies than any other solution. And it's cross-platform–available for your PC users, too. It's easy to install and manage and simple for your employees to understand and use.

Using Now Up-to-Date & Contact you can schedule meetings for multiple users, view multiple, simultaneous calendars, and reserve rooms and resources. You can share contact information about your customers, prospects and vendors. And using our free server software you can set it up in minutes and share with users in the office or from anywhere with an internet connection.

Call us now at 866-527-0556 or email us at mactech@nowsoftware.com and we'll send you our free evaluation kit, including the book that will make it all easy, "Take Control of Now Up-to-Date & Contact" from Take Control books!

NOW SOFTWARE

Phone: 866-527-0556

Web: www.nowsoftware.com

Objective C. Setting up Outlets, Targets and Actions in Python is very easy, and Interface Builder will work nicely with your Python versions of these Cocoa concepts.

Move ahead to "Defining the Controller Class" in the Apple tutorial. Choose File > New File in XCode. In the left pane of the New File dialog, select Pure Python, and create a new Python Module file as we did earlier, this time calling it ConverterController.py. Note that there are templates for some Python classes listed under the Cocoa section of the New File dialog, but since Python is so succinct, those templates don't do a whole heck of a lot for you. Feel free to use them, we're doing everything here from scratch partly for the learning experience, and partly because the Python Cocoa templates are really only giving you three or four lines of easy-to-type code.

Listing 2: ConverterController.py

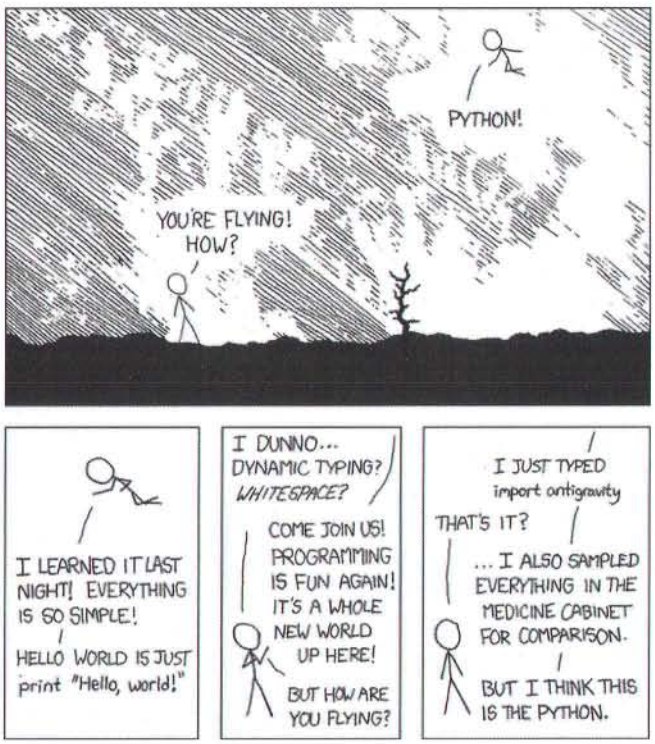
Currency Converter Controller

This is our view controller. Note the three IBOutlet properties, and the IBAction method. These are automatically detected by Interface Builder so that we can connect our UI directly to the Python code.

```
#
#  ConverterController.py
#  Python Currency Converter
#

from Foundation import NSObject, NSWindowController
from objc import IBOutlet, IBAction

from Converter import Converter

class ConverterController(NSWindowController):
    amountField = IBOutlet()
    dollarField = IBOutlet()
    rateField = IBOutlet()

    def awakeFromNib(self):
        print "awake!"

    @IBAction
    def convert_(self, sender):
        converter = Converter()
        converter.sourceCurrencyAmount =
self.dollarField.floatValue()
        converter.rate = self.rateField.floatValue()
        amount = converter.convertCurrency()
        self.amountField.setFloatValue_(amount)
        self.rateField.selectText_(sender)
```

There are three Outlets needed in our ConverterController class. As you see in Listing 2, we can create outlets simply by instantiating the PyObjC class IBOutlet, and saving off the outlet as a member variable in our class.

We create one IBOutlet for each field in our user interface: amountField, dollarField, and rateField. We assign each of those to member variables of the ConverterController class. If you have some experience with Python, you might be a bit chagrined by the way the Outlets are initialized in the class. They are declared directly as attributes of the class, and initialized when the module is first loaded in, rather than being initialized in the __init__ method. This is intentional - the IBOutlet initializer is just a marker for Interface Builder, and it



Comic courtesy www.xkcd.com

needs to be a part of the class before it is instantiated, so that is why the Outlets are initialized this way. If you instead moved those initializations into the __init__ method, Interface Builder wouldn't find those Outlets.

We can also set up an Action in ConverterController. An action is just a method on our controller class, but we need to mark it as an action so that Interface Builder knows we intend it to be an action. In Python, we use a method decorator to mark a method as an action. In Listing 2, notice the line above the convert method, "@IBAction". That syntax might look a bit odd (particularly since the @ symbol has a different meaning in Objective C than it does in Python), but Python allows you to add little enhancements to methods via this syntax. @IBAction is just being used here as a tag to alert Interface Builder that this method is an action.

Also note that the convert_ method has an underscore after it. That's not a typo or an odd habit, but instead it's a convention for converting Cocoa Objective C method names to Python method names. An Objective C method name is usually split up into sections, one section for each parameter passed to the method. There is a colon after each part of the Objective C method name. Python uses more traditional method names, followed by a parameter list. To rewrite an Objective C method name as a Python method name, the convention used in PyOBjC is to simply replace the colons with underscores, and string the whole method name together. Thus, an Objective C method called setValue:forKey: becomes setValue_forKey_ in Python. Thus, the Objective C method convert: becomes the python method convert_. If an Objective C method doesn't accept any parameters, then the Python method name will be

the same as the Objective C method name (e.g. [myString capitalizeString] in Objective C becomes myString.capitalizeString() in Python).

Great! Now we're at the section of the Apple tutorial titled "Interconnecting the Controller with the View". Let's do it! Will it work? Sure!

Go back to the Interface Builder - your xib file should still be loaded (if not, double-click MainMenu.xib in the Python Currency Converter XCode project to reload it). Select File > Read Class Files in Interface Builder. A file chooser will appear. Select ConverterController.py. If there are any major typos in ConverterController.py, Interface Builder will tell you that it can't parse the file. Check it out, make sure there aren't any tabs-vs-spaces problems, and try again.

Drag an Object from the library to the MainMenu (English) window. Select the new object, and navigate to the Identity tab in the Inspector, as described in the Apple tutorial. In the Class drop-down box, and select ConverterController from the drop-down menu... it will be there! (Yes, Interface Builder found it!) Our convert: method is now showing in the Class Actions section. Also, our three Outlets, amountField, dollarField, and rateField, are visible in the Class Outlets section.

Hooray, we can now wire up our interface!

Control-drag a connection from the ConverterController object to the Exchange Rate field, and choose rateField from the window that pops up. Wire up dollarField and amountField the same way. Too easy!

Now control-drag from the Convert button to the ConverterController object to wire up the convert: action. Yeah!

We can double-check all of the connections by clicking on the inspector's Connections tab for the ConverterController object. Mouse over the outlets and the action shown there to see the corresponding user interface element highlighted. Fix up anything that isn't connected correctly.

Now we're going to connect our ConverterController with the model we created in Convert.py. Open up ConverterController.py in XCode. Add a new import line at the top that says "from Converter import Converter" - what this means is, import the class Converter from the module/file named Converter.

Now we're going to add a line to ConverterController that will instantiate a Converter class for us. See the line "self.converter = Converter()" in Listing 2. The Converter is instantiated in the ConverterController's __init__ method, so it will be initialized when the ConverterController is created.

Now we can make our convert_ method actually do something!

Hooking Up Our Model

We'll follow the Apple tutorial's approach of creating a new Convert() object each time we click on the Convert button. The Convert object will be garbage collected by Python once we're done with it.

Look at Listing 3 to see how we create and use the Convert object in the convert_ method. The convert_ method is almost line-by-line identical to the convert: method from Apple's tutorial. We convert the Objective C message sending syntax into the Python method invocation dot-syntax, and everything works just as we would expect.

Yeah! That was too easy!

Tabs vs Spaces Reminder

We all know that programming involves a lot of indenting, and we all know that tabs can be used to indent, as can spaces. Inconsistency in tabs is usually no more than an annoyance, but in Python, it will cause an error. Whitespace is meaningful in Python, so if you use both tabs and spaces, the Python compiler will not know exactly what you mean.

XCode presents us with a bit of confusion right off the bat. When you ask XCode to create a new Python file for you, XCode will use four-space indentation. However, XCode's editor defaults to using real, actual tabs for indentation. You can change the XCode editor setting to use spaces instead of tabs, but since you can't set a file-type specific indentation preference, nor can you ask XCode to automatically determine the indentation style used by a file, it's all kind of a huge pain. Perhaps XCode will have a more flexible tab/spaces preference in the future.

The Python files that XCode has already generated for our project (such as `Python_Currency_ConverterAppDelegate.py`) use four spaces to indent. And, quite a few python scripts out in the wild use four spaces. It's easy to set XCode to use spaces, not tabs, to indent. Select XCode > Preferences > Indentation and un-check "Tab key inserts tabs, not spaces". If you really want to leave your XCode setting on tabs instead of spaces, you could go the other way and change the XCode-created files to be tab-indented instead of space-indented, it's up to you.

If you do accidentally mix tabs and spaces, it's no big deal. Lucky for us, one of Python's strengths is that it tells you exactly what went wrong, exactly where, and exactly why. If you have tabs and spaces mixed, you will get an Indentation Error that tells you the first line in the file that does not match the indentation convention of the earlier lines in that file.

REALbasic 2008

Cross-Platform that Really Works.*

Create your own software. Easily.

REALbasic is the powerful, easy-to-use tool for creating your own software. At REAL Software, we call it a problem solver. You've probably said, "Wouldn't it be great if there was an application that could ..." REALbasic fills that blank.

REALbasic compiles native applications for Mac OS X, Windows and Linux from one set of source code. Each version of your software looks and works just as it should in each environment. You can even create a Universal Binary with one mouse-click.

Try REALbasic risk-free!

REALbasic includes a 90-day money back guarantee.

www.realbasic2008.com

Troubleshooting

BUT WAIT, you say, IT ISN'T WORKING FOR ME MAN!!!

There are a lot of things that can go wrong here. XCode isn't responsible for compiling your Python scripts, so errors won't show up until you try to run your code. Finding out problems at run-time is fine, and very informative, but for simple things like typos, it would be nice to have your editor tell you those things.

There are great tools, like pylint, that can do this type of checking for you. And there are great IDEs, like Eclipse's PyDev, that integrate pylint into the editor, doing real-time checks of your code as you edit it. XCode does not yet have these cool features for Python, so there is a big chance you will have mistyped something and not know about it until runtime.

Look in the Console window! If something went wrong in Python, there will be a traceback in your console window. It will usually point to exactly the spot where things went wrong. There is a handy Preferences setting in XCode, under Debugging, that will open the console every time you start debugging. It's worth turning that on (look for the "On Start" drop-down menu).

Occasionally, something will go wrong deep down in the PyObjC bridge, and you'll se a dialog box pop up with an error message. Don't worry, there should still be a traceback in your Console window pointing right at the problem. If all else fails, you can sprinkle your code with "print" statements to see how far your code is actually getting.

Python – It's What Your Brain Craves

Enjoy! This is just the tip of the iceberg, you can do quite a bit with Cocoa-Python Applications in XCode. If you are an Objective C die-hard, you might be looking askance at this whole idea, wondering why you would ever want to learn a new language just to do what you can already do in Objective C. But if you've worked with Python before, or if you've paid attention to how little code it took for us to get things going here, you'll know why all of those Objective C programmers should give Python a shot. It makes programming fun again!!!!

About The Author

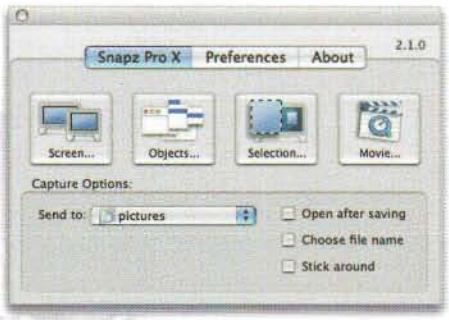
Scott has been a Mac software developer since he got his fist Powerbook in 1991. Scott has written many applications for many platforms, primarily in the video game industry. He currently holds the title of Director at Wideload Games in Chicago, and has developed and published AcidSolitaire Collection for iPhone via his own company, Red Mercury, LLC.

SnapzProX

Free Trial Version!

If a picture is worth a thousand words, imagine how priceless a movie would be...

Snapz Pro X allows you to effortlessly record anything on your screen, saving it as a QuickTime® movie that can be emailed, put up on the web, or distributed however you want.



Why take a static screenshot when Snapz Pro X makes creating a movie just as easy?
In addition to a video capture engine that is 20 times faster than anything else on the market, Snapz Pro X has so many other new features, you'll quickly wonder how you ever lived without it!

Download your free trial now:
http://www.AmbrosiaSW.com/MacTech

AMBROSIA® SOFTWARE INC

2004 winner macworld eddys

Macworld

Mac

Universal

Snapz Pro X requires Mac OS X 10.3.9 or later. Snapz Pro X, Ambrosia Software, Inc., and the Ambrosia Software logo are registered trademarks of Ambrosia Software, Inc. QuickTime is a registered trademark of Apple Inc.

Torque it to the Macs

Get up and running with the Torque Game Engine

by Kenneth C. Finney

Back Story

Way back at the turn of the century, four intrepid adventurers departed from the clutches of a smothering oligarchy in search of freedom, broader horizons, and a decent latté. They took with them, cleverly ensorcelled from the oligarchy's wizards, a set of incantations... Okay, this is just getting to be too darned silly.

Serious face.

The Torque Game Engine began life as the engine for AAA game Tribes 2, made by Dynamix, and released back in 2001. When Jeff, Rick, Tim, and Mark decided to leave Sierra (who by then owned Dynamix) and start over again, they did it under two unusual sets of circumstances.

Circumstance, the first, was that they somehow managed to persuade the bosses at Sierra to let them take the I.P. for Tribes 2 with them. As amazing as that seems, their reason for doing that led to Circumstance, the second. They tidied up the source code, and then hung out their shingle:

For Sale: Torque Game Engine, $99. Source Code included.

Ninety-nine dollars? Ninety-nine dollars!! *Source Code included!* Holy hyperbole!

Not only was the source code included, so was sample art: models, audio, levels, terrain. The mind boggled. Okay, my hyperbole generator is starting to make a rattling noise, so I'll stop.

This was the engine behind Tribes 2, and not merely the result of a basement roll-your-own team, or a computer science capstone project. In those heady early days, those of us that were involved spent a great deal of time poring and puzzling over C++ and TorqueScript code, for there was nothing in the way of documentation. And it even had a different name then: V12. It took me months to get used to the name change to Torque.

But those guys had a vision, and they went after it with a vengeance. There are so many stories to tell and so little space available. One of those stories is the "code once, run anywhere" story, and that's where this article comes in. All of the code in this article was written and tested on that Other System, before copying it over to the Macintosh for testing. And nothing in the code (or the art assets) was changed *after* copying over, either.

Of course with Torque technology, you can create First Person Shooters (FPS). The code and sample art to start doing that is available when you buy the license. You also get code and sample resources for a racing game, again right out-of-the-box. There are additional resources and code available for things like flying vehicles, weather simulation, and so on. You can, with a low cost extra "bolt-on" kit, get a leg-up on creating multiplayer Real-Time Strategy (RTS) with the RTS Starter Kit. If 2D is your bag, the Torque Game Builder (TGB), which includes Torque2D, is a separate purchase–but still very low cost. It provides samples and source for side scrolling platformers, 2D shooters, multiplayer checkers, and so on.

A massive online development community (www.garagegames.com) provides free modifications yips and code resources that can be applied to the core engine, for everything from watercraft support to full-screen shader effects.

The development team at TubettiWorld Games has used Torque to deliver a 3D-Racing game for a major toy manufacturer. We've also used it in the past for Serious Games training applications in the Nuclear, Medical and Defense industries. It's a very versatile tool.

Preparation

In this article we will be concentrating on the game control scripts found in the game sub-tree. To prepare for this, you need to set up your development tree, as follows:

Browse your way to www.garagegames.com

Click on **Products**, then scroll down and select **Torque Game Engine**.

On the right, click on the button that says: **Download TGE DEMO**.

You will be presented with a page that asks for your name and email. This is purely optional, as is the checkbox for mailings. Fill these out (or don't) however you like, and then click on **Begin Download**.

Click on the Download Now button below the blue apple and description of the Macintosh Demo. Save it to wherever you want to save it.

Extract the demo into a folder on a volume of your choosing.

You will now have a folder called `Torque Game Engine 1.5.2 Demo` located wherever you extracted it to. Dive inside there. You'll find a folder called **Torque Demo**. Make a copy of that, and rename the copy to `MacinTorque`.

You should do all of your work in the MacinTorque folder.

You probably won't use more than an additional 15MB of disk space for the download, 30MB more for the extraction, and 30MB more for the copy for a total of 75MB, but you should have more available for backups and temporary files and so on.

In the MacinTorque folder, next to the Torque Demo OSX executable, and the command files, and the ReadMe, you will find a file called main.cs. Delete it. Make sure you have the right one–there are several located in other folders. Also, delete the folders show, demo and creator. Don't worry–you still have the originals, but don't delete anything else. Now for the rest:

Go to ftp.mactech.com/src/mactech/volume25_2009/25.02.sit, open the file, and copy the MacinTorque_1.zip archive to a location of your choosing.

Reach inside the freshly downloaded MacinTorque_2.zip file and grab the folder called game, and drag it into the MacinTorque folder you created inside the Torque Game Engine 1.5.2 Demo folder. It's okay; you've already deleted conflicting files and folders, so everything will be fine.

What you will end up with is the support code for this articles project. There is still more code to be added, but that comes later.

The folder structure in the game control folder has been deliberately chosen to *not* reflect the folder hierarchy used by GarageGames. This is in order for you to learn that there is very little in the way of *required* folder structure for Torque to depend on. It also makes it much easier for the uninitiated to find things. GarageGames likes to, for good reason, keep the client code in one branch of the game control tree, server code in a different branch, and finally, data in a third branch. In our little example, we are only going to isolate the data in its own folder tree. Even then, I'm keeping the player model files in the game folder adjacent to the code that uses it.

While we are talking about folders and such, a word about paths. TorqueScript conforms to Linux Unix style of paths, which means right-leaning slashes (/). The dot (.) symbol refers to the current folder, and that translates to meaning whatever folder the script that is currently executing is located in. The double-dot (..) means the folder above the current folder. And finally, the tilde (~) operator, when encountered at the start of a path, means the currently active package. There will be more about packages later on. Oh yeah–the tilde is also used when in game to invoke the script console, but don't tell anyone I told you that.

The code in MacinTorque is very close to the bare minimum in terms of the game control code. In later articles we will expand on this skeletal implementation as we add more and more useful features and flesh out the game.

Root Main Module

When the Torque engine is launched, one of the first things it does is look for a script module called main.cs located in the same folder as the executable.

Once it has found the root main module, Torque compiles it into memory as a special binary version containing *byte code*, a machine-readable format. Torque does the same thing with all other script modules, but with them it also writes their byte code out to disk in the form of a file with the extension *dso*. The root main module is the only script module that does not have its byte code written to disk.

After it has loaded the root main module, the game engine then begins executing the instructions in the module. The root main module can be used to do anything you like, but the convention established with the GarageGames code is that it does things like command line parsing, usage help, and loading *packages* or *add-ons* (also sometimes called *mods*).

So here's what I want you to do. Using your favorite text or programming editor, create a blank file in the MacinTorque folder, right there next to the Torque Demo OSX executable. Then type in the following code, and save it. Of course, whenever I give you code to type in, only type in the stuff that's in Courier font, don't enter the function name header in Palatino or the comment that follows it.

Also, make sure to note the subtle difference between parentheses–() and braces–{}. For example, the first thing you should be typing in the main.cs file is the function definition code for the OnStart() function. After the words function OnStart there are two parentheses–opening and closing, and then two braces (some people call them "curly brackets". Ask them, not me...), opening and closing. For you old C-hands, please be patient. There once was a time when you got them mixed-up too, you know.

MacinTorque/main.cs

The following code is in-line in the main.cs script file, as written, and not within any function

```
function OnStart(){}
function OnExit(){}
SetLogMode(6);
SetModPaths("game;common");
Exec("common/main.cs");
Exec("game/main.cs");
OnStart();
```

The functions OnExit() and OnStart() are *stub routines*–functions that are defined but actually do nothing. The common code modules have a call to each of these routines, but we have nothing for them to do. We could just leave them out, but it's good practice is to provide stubs to prevent pointless warning messages from cluttering up our log file; often when the Torque engine tries to call a nonexistent function, it generates a warning.

After that we set the log mode. Log mode 6 means that the file is opened for each log message, and then closed after the message has been written. It also means that the log file is overwritten each time Torque is run.

Next, we build the list that helps Torque find our add-ons. We notify Torque about the required folder paths by passing the list as a string to the SetModPaths() function.

Then we call the main module for the common code. This will proceed to load all the required common code modules into memory, initialize the common functions, and basically get the ball rolling over there. We will talk about the common code modules provide default functionality for.

After that we do the same thing for the game control code modules, the details of which we will cover later. For now, it's enough to know that the game folder is where scripts and art assets specific to our game are located.

Then we actually start loading the add-ons using the previously defined `LoadAddOns()` function.

Next we use the Exec function to load and execute the add-on packages in order of their appearance in the string literal we used in `SetModPaths()`, with the `main` module in the `common` package being first and the `main` module in the `game` package next. The Exec function will execute any in-line code it encounters in a module, and then load any functions or datablocks (structures) it encounters into memory for later execution.

At the end of the root `main` module is a call to `OnStart()`, which provides a good opportunity to explain how packages work in Torque. If you look in `common/main.cs` you will see a definition of `OnStart()`. You'll also find a definition of `OnStart()` in `game/main.cs`. Or rather, you would if you jumped ahead in the article–we're going to be putting one there as well. And finally, there's that definition of `OnStart()` that we'd made there at the top of the root `main.cs`.

So now you are wondering: which `OnStart()` gets called by that last line in the root `main.cs`? Well, the ordering of the package activation decides that. You'll notice that `common/main.cs` is Exec'd before `game/main.cs`. Both of those modules activate their respective packages, `common` and `game` according to an order of precedence determined by the order in which they were activated. The last activated package overrides functions with the same names in earlier activated packages. The first activated package is considered to be the parent of the second activated package.

If you want to call a function in a parent package, you can do so by prepending `Parent::` in front of the function call.

So in our case here, the order of activation (last is first) dictates that the `OnStart()` in `game/main.cs` is called first. And this version just happens to immediately place a call to `Parent::OnStart()` that is located in the common package. And *that* version calls its parent, which turns out to be the stub routine we created at the top of the root `main.cs`! It might seem to be a bit complex, but it provides a convenient and powerful mechanism for just dropping in new scripts on top of existing scripts and just having them work without modifying the original functionality.

Now, when we get to the end of the module, the various threads initiated by the `OnStart()` calls are ticking over, doing their own things.

So then what? Well, our next point of interest is the `game/main.cs` module, which we loaded with the Exec function just before we called `OnStart()`.

Game Main Module

The `game/main.cs` module for the control code is next on our tour. Its primary purposes in MacinTorque are to define the

JUST® mobile

Xtand™

iPhone™ not included

Xtand. Stands Above The Rest.

HIGHLY RECOMMENDED
iLounge.com | All Things iPod

Macworld Conference & Expo | Booth No. 4320 January 5-9,2009

www.xtand.net

game control package and to call the game code initialization functions. Type the following code into a new file, and save it as `MacinTorque/control/main.cs`.

MacinTorque/game/main.cs

Global variables that need to be initialized for use by the engine or the common code base. These are inline, and not in any function.

```
$Pref::Server::MaxPlayers = 64;
$pref::Video::allowOpenGL   = true;
$pref::Video::displayDevice = "OpenGL";
```

Game

Package that exercises control over the game activities.

```
package game {
```

OnStart

Called by root main when package is loaded

```
    function OnStart(){
    Parent::OnStart();
    InitBaseServer();
    Exec("~/server.cs");
    CreateServer("SinglePlayer",
"game/data/maps/mactech1.mis");
    InitBaseClient();
    InitCanvas("MacinTorque");
    Exec("~/client.cs");
    %connection = new GameConnection(ServerConnection);
    %connection.ConnectLocal();
}
```

OnExit

Called by root main when package is unloaded

```
    function OnExit(){
       Parent::onExit();
    }
```

(inline)

Tell Torque to activate the game package. These are inline, and not inside any function.

```
}; // game package
ActivatePackage(game);
```

There's a lot happening here, relatively speaking, so let's dive right in.

First we have three global variables defined. You need to realize that Torque is typeless and doesn't require forward declarations. So these assignments, being the first time Torque encounters them in script, are de facto declarations as well.

The dollar sign ($) prefix tells us that these are global variables. The `Pref::` part is like a poor man's namespace descriptor–in fact so is the `Server::` part. Also notice that while typeless, Torque treats data according to context. There are numerics (`64`), tokens (`true`), and strings(`"OpenGL"`), and we can see examples of all three in the three variable assignments. So watch out when you create your own variables–you need to remember how you intended to use them. You also need to be very careful about spelling errors. If you mistype a variable name, most likely Torque will just create a new variable in the symbol table, and merrily continue on doing the wrong thing...

The `$Pref::Server::MaxPlayers` value of 64 is just a default setting. On modern computers, I have had over 130 players connected to a single server with no noticeable penalty. It depends on bandwidth utilization, and how much actual work the server has to do, and other game design decisions. For a straightforward first person shooter, you should encounter no problems hosting more than 64 players.

Obviously, because we are running on a Macintosh, we want to use OpenGL, so the next two variables are necessary to make that happen.

Next we encounter the package declaration for the `game` package. All of the code in the ensuing code block (including all of the code in any scripts Exec'd while in the code block), becomes part of the `game` package.

And then we find the definition of the game package's `OnStart()` function. The first thing *it* does is call the *parent* `OnStart()` function, which is the one in the `common` code base. By calling the parent first, we can then later override anything the parent has done, if we need to.

Following the call to `InitBaseServer()`, which sets up the basic server features that are already created for us in the `common` code base modules, we load the `game/server.cs` module containing server side control code, which we'll look at a little later. Aha! You think you've got me! It has `~/server.cs` in the code, you're thinking, and that's not what I wrote. You're right. The tilde is the stand-in for the *current active package* and that happens to be `game`. I was just translating for those who are new to this stuff.

So, next we create the server threads, with the call to `CreateServer()`. We tell it to start in single player mode (don't solicit external connections, and don't contact any master server), and also tell it to load a mission file: `game/data/maps/mactech1.mis`. Notice that the tilde is not used in place of the word game in the mission file path. It doesn't work in this usage. I don't know why.

The client side mirrors the server side a little. First we set up the basic client features by calling `InitBaseClient()`. Next we have to prepare the actual graphics device object, which we call the `Canvas`. Then we load the client-specific support code using the `Exec` statement, and finally, we open up a connection to the server side using the same technique we would have used if we were connecting to a server in Sarnia. Note that a handle to the `GameConnection` object that we instantiate with the `new` statement is saved in the local variable `%connection` so that it can be used in the next statements. The percent sign prefix (%) indicates that the variable is a *local* variable and is only in scope inside the function.

Finally, there is the `OnExit()` function, which does nothing more than pass the buck to the parent `OnExit()` function in the `common` main module.

Now while it might *seem* that these versions of `OnStart()` and `OnExit()` are the only functions in the game package, I should remind you that all of the functions in the Exec'd files in the `OnStart()` function are *also* in the game package. As are the files Exec'd inside those files. And so on...

The game package is activated by the very last call: `ActivatePackage()` which is used as an in-line statement in this case.

Server

The `game/server.cs` module is where game-specific server code is located. Most of the functionality that is carried in

this module is found in the form of methods for the GameConnection class. Type in the following code, and save it as MacinTorque/game/server.cs.

MacinTorque/game/server.cs

Provide client connection management, player handling, and most game logic.

OnServerCreated

Once the engine has fired up the server, this function is called by the engine. It calls the module that maps the animation sequence files to the animation names, then creates an instance of a PlayerData datablock.

```
function OnServerCreated(){
   Exec("~/animations.cs");
   datablock PlayerData(MaleAvatar){
      shapeFile = "~/player.dts";
      cameraMaxDist = 3;
   };
}
```

GameConnection::OnClientEnterGame

Extension to the GameConnection class to handle player attaching to the server. Called by engine.

```
function GameConnection::OnClientEnterGame(%this){
   %this.spawnPlayer();
}
```

GameConnection::SpawnPlayer

Extension to the GameConnection class to handle player spawning issues.

```
function GameConnection::SpawnPlayer(%this){
   %this.createPlayer("0 0 120 1 0 0 0");
}
```

GameConnection::CreatePlayer

Extension to the GameConnection class to assign a new avatar object to the attached player via the game connection. It creates the player avatar, and gives control to the player's client.

```
function GameConnection::CreatePlayer(%this, %spawnPoint){
   if (%this.player > 0)
      Error( "Attempting to create an angus ghost!" );
   }
   %player = new Player() {
      dataBlock = MaleAvatar;
      client = %this;
   };
   %player.setTransform(%spawnPoint);
   %this.player = %player;
   %this.setControlObject(%player);
}
```

(inline)

Stub routines. These are inline, and not inside any function.

```
function ClearCenterPrintAll(){}
function ClearBottomPrintAll(){}
```

The first function, OnServerCreated, manages what happens immediately after the server is up and running. In our case we need the player-avatar datablocks and methods to be loaded up so they can be transmitted to the client.

The datablock used is the PlayerData class, and its instance has a name: MaleAvatar. It is a very minimal datablock definition; many more properties than we need right now are available for an avatar of the PlayerData class. In fact, PlayerData datablocks are often 50 to 100 lines of code in size! By going with a minimal definition, we are leaving the rest of the formal PlayerData properties to their default settings.

Of course, we need to tell the engine which model to use, and in this case, the model is named `player.dts`. *Dts* files are Torque's native shape format. They're created by a model exporter that is run inside the application that was used to make the model. In this case,It so happens that Milkshape on Windows XP was used to create the model we are using. There are exporters available from GarageGames for many other 3D tools, including but not limited to: 3D Studio Max, Maya, Blender and Lightwave.

We can also define our own properties for the datablock and access them through an instance of this datablock from anywhere in the scripts. Datablocks act something like templates with read-only initial values. Datablocks are sent from the server to all new clients when they connect, and their fields cannot be changed at runtime.

The As I mentioned, the datablock refers to a model called `player.dts`; this is a stickman-like character that we use at TubettiWorld Games as a stand-in for any yet-to-be modeled character. Oh, and by the way, his name is Stick Norris.

Next we define some `GameConnection` methods. The first one, `OnClientEnterGame`, simply calls the `SpawnPlayer` method, which then calls the `CreatePlayer` method using the hard-coded transform provided. The first three terms of the transform are the XYZ coordinates, and the last four are the rotation vector. So the player is spawned at an altitude of 120 World Units (W.U.–about the same as a metre). This is about 20 W.U. above the ground. He is also facing straight up the positive X axis (often used as North).

`CreatePlayer` then creates a new player object using the MaleAvatar datablock defined earlier. It next applies the transform (which we created manually earlier) to the player's avatar and then transfers control to the player. This means that any inputs on this particular client (accessible via the `GameConnection`) are applied to this particular player character.

Finally, there are a couple more stub routines of no particular import to us at the moment. But we do want to keep that console log tidy.

Animations

Torque supports animated models using two different techniques:
embedded animations and animation sequence files.

Embedded animations are the form most familiar to animators, where the animation sequences are contained with the 3D model file itself. Embedded animations are either skeleton (bones) based (where bones are positioned for each key frame, and the mesh follows along), or *morphs* (where the mesh itself is changed at each key frame). Each animation sequence bears the name of an animation supported by the engine, and can then be called either from the engine or from script.

Animation sequence files (`.dsq`) are exported to the engine separately from the model, and are only skeleton-based. Keeping them separate allows any other model to re-use the same animations, as long as the skeleton is near enough to the original in shape, and that the bones have the same names. For its *Kork the Torque Orc* model, GarageGames uses the Biped skeleton that

Macworld
Editors' Choice
★★★★★

Aquafadas

BannerZest

Flash Banner - Flash Animations - Coverflows

Create fully animated professional banners to spice up your website or blog

Discover also on
www.aquafadas.com
Get the most of your digital life !

PulpMotion SnapFlow VideoPier iDive Ave!Comics

is pretty well a standard skeleton that comes built-in to many modeling tools, like 3D Studio Max. You aren't required to use the Biped skeleton yourself, unless you intend to take advantage of the animation sequence files provided by GarageGames.

The animations we are using here are a subset of the ones included in the stock animation sequence files that GarageGames includes with its demo.

In the module `game/server.cs`, in the `OnServerCreated()` handler, we loaded `game/animations.cs` prior to defining the player character's datablock. Let's create that file, then poke around in it a bit.

Type the following code into an new empty file and save it as `MacinTorque/game/animations.cs`.

(inline)

These are inline, and not in any function.

```
datablock TSShapeConstructor(PlayerDts)
{
   baseShape = "~/player.dts";
   sequence0 = "~/animations/player_root.dsq root";
   sequence1 = "~/animations/player_forward.dsq run";
   sequence2 = "~/animations/player_back.dsq back";
   sequence3 = "~/animations/player_side.dsq side";
   sequence4 = "~/animations/player_lookde.dsq look";
   sequence5 = "~/animations/player_head.dsq head";
   sequence6 = "~/animations/player_fall.dsq fall";
   sequence7 = "~/animations/player_land.dsq land";
   sequence8 = "~/animations/player_jump.dsq jump";
   sequence9 = "~/animations/player_sitting.dsq sitting";
   sequence10 = "~/animations/player_standjump.dsq standjump";
};
```

It's a datablock of the type `TSShapeConstructor`, and it's been given the name `PlayerDts`. The base shape referred to is the exact same file that the `PlayerData` datablock in the `game/server.cs` module. It doesn't need to be, however. It just needs to have a suitably structured skeleton with properly named bones.

The rest of the lines the datablock are a series statements that map an animation sequence file to an animation name. Inside the double-quoted string literals there are actually two strings: the first string is the path to the sequence file, and then separated by a space is the actual animation name. All of the animation names used here are already understood by Torque, and are used internally. You can use your own animation names, even if Torque doesn't know about them internally, by calling the animations from script. But that's a nice topic for another article.

Note that the field names–sequence0, sequence1, and so on–must appear in sorted order if you want them to work properly. So they are numbered *and* ordered, just as if they were issued from the Department of Redundancy Department.

Client

The `game/client.cs` module is over-full of interesting things. This is another module that will need to have some of its code distributed elsewhere if the project grows to be more extensive. The main activities taking place in here are as follows:

* Creation of a key map with key bindings
* Definition of a callback that gets called from Torque to generate a 3D view
* Definition of an interface to hold the 3D view
* Definition of a series of functions that hook key commands to avatar motion
* A series of stub routines

Here is the `game/client.cs` module. Type it in, and save it as MacinTorque/game/client.cs.

MacinTorque/game/client.cs

Everything that pertains to user input or display is handled by this module.

(inline)

These are inline, and not in any function.

```
new ActionMap(playerKeymap);
new GameTSCtrl(PlayerInterface) {
   profile = "GuiContentProfile";
   noCursor = "1";
};
```

PlayerInterface::onWake

```
function PlayerInterface::onWake(%this){
   activateDirectInput();
   playerKeymap.push();
}
```

GameConnection::InitialControlSet

```
function GameConnection::InitialControlSet(%this){
   Canvas.SetContent(PlayerInterface);
}
```

GoLeft

```
function GoLeft(%val){
   $mvLeftAction = %val;
}
```

GoRight

```
function GoRight(%val){
   $mvRightAction = %val;
}
```

GoAhead

```
function GoAhead(%val){
   $mvForwardAction = %val;
}
```

BackUp

```
function BackUp(%val){
   $mvBackwardAction = %val;
}
```

DoYaw

```
function DoYaw(%val){
   $mvYaw += %val * 0.02;
}
```

DoPitch

```
function DoPitch(%val){
   $mvPitch += %val * 0.02;
}
```

DoJump

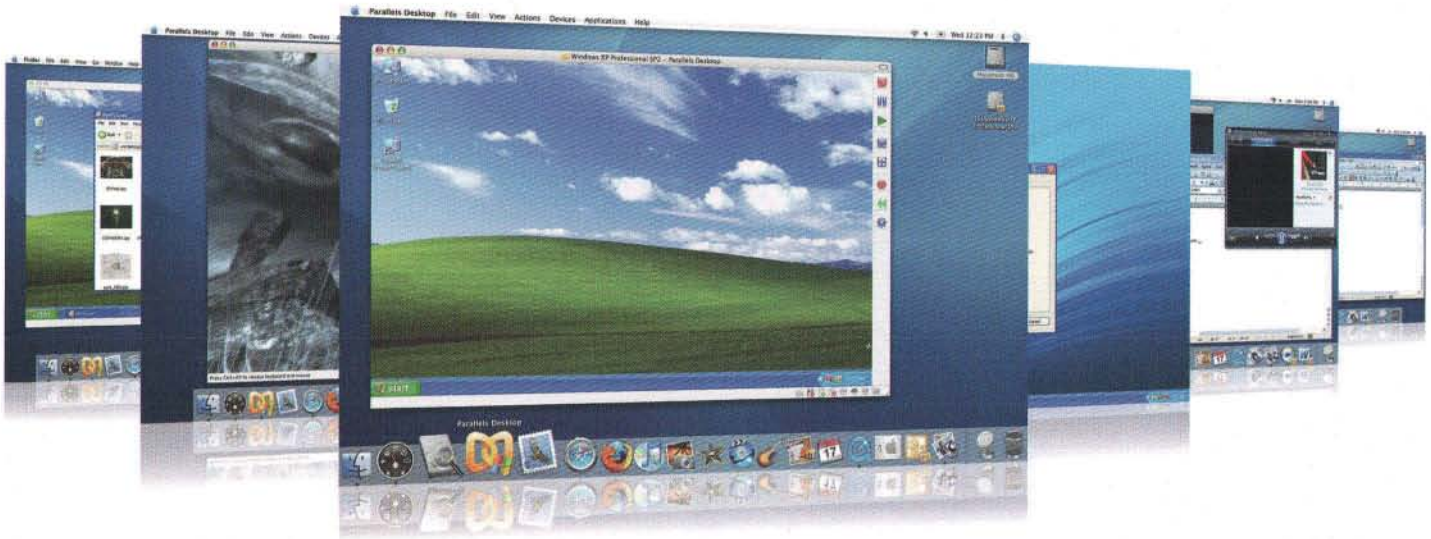
```
function DoJump(%val){
   $mvTriggerCount2++;
}
```

Toggle3rdPPOVLook

```
function Toggle3rdPPOVLook( %val ){
   if ( %val )
      $mvFreeLook = true;
```

Plays well with others.

Run Windows and Mac OS X at the same time - without rebooting.



With Parallels® Desktop 3.0 for Mac, you can:

- Automatically open Windows files with Mac software and Mac files with Windows software
- Bulletproof your virtual machine from Windows system crashes and malware
- Run selected Windows-only 3D games and applications with 3D graphics support
- Browse through Windows files and folders without launching Windows
- Move your existing Windows PC to a Mac without losing data or reinstalling any software
- Completely share files and folders between Windows and Mac

Parallels™

Parallels® Desktop 3.0 for Mac provides even more ground breaking features and capabilities than previous versions. Already The #1 Selling Mac System Utility according to NPD Techworld (09/06 – 12/07) and recently named an InfoWorld Magazine "2008 Technology of the Year", Parallels® Desktop is trusted by over 800,000 users world wide.

Want better Windows and Mac integration? Want to run the hottest PC games and graphics software? Worried about security and system mishaps? Parallels® Desktop 3.0 for Mac includes 50+ new features and enhancements, including a number of integration features and virtual machine utilities unavailable anywhere else.

To discover why Parallels® Desktop is the leading desktop virtualization solution for running Windows on a Mac, visit us online or call us at 1 (425) 282-6405.

PC WORLD 100 BEST PRODUCTS of 2007

Macworld BEST OF SHOW 2007

Download a trial today at www.parallels.com/download

Website – www.parallels.com Email – sales@parallels.com Phone – 1 (425) 282-6405

```
   else
      $mvFreeLook = false;
}
```

Toggle1stPPOV

```
function Toggle1stPPOV(%val){
   if (%val){
      ServerConnection.setFirstPerson($firstPerson);
      $firstPerson = !$firstPerson;
   }
}
```

(inline)

Function calls. These are inline, and not in any function.

```
PlayerKeymap.Bind( mouse, button0, MouseAction ); // left
mouse button
PlayerKeymap.Bind(keyboard, w, GoAhead);
PlayerKeymap.Bind(keyboard, s, BackUp);
PlayerKeymap.Bind(keyboard, a, GoLeft);
PlayerKeymap.Bind(keyboard, d, GoRight);
PlayerKeymap.Bind(keyboard, space, DoJump );
PlayerKeymap.Bind(keyboard, z, Toggle3rdPPOVLook );
PlayerKeymap.Bind(keyboard, tab, Toggle1stPPOV );
PlayerKeymap.Bind(mouse, xaxis, DoYaw );
PlayerKeymap.Bind(mouse, yaxis, DoPitch );
GlobalActionMap.BindCmd(keyboard, escape, "", "quit();");
GlobalActionMap.Bind(keyboard, tilde, ToggleConsole);
```

(inline)

Stub routines. These are inline, and not in any function.

```
function onServerMessage(){}
function onMissionDownloadPhase1(){}
function onPhase1Progress(){}
function onPhase1Complete(){}
function onMissionDownloadPhase2(){}
function onPhase2Progress(){}
function onPhase2Complete(){}
function onPhase3Progress(){}
function onPhase3Complete(){}
function onMissionDownloadComplete(){}
```

Right off the bat, a new ActionMap called playerKeymap is created. This is a structure that holds the mapping of key commands to functions that will be performed–a mechanism often called *key binding*, or *key mapping*. We create the new ActionMap with the intent to populate it later in the module.

Then we define the 3D control (TS, or *ThreeSpace*) we call `PlayerInterface` (because that's what it is), which will contain our view into the 3D world. It's not a complex definition. It basically uses a profile defined in the common code–something we'll explore in a later article. If we want to use our mouse to provide view manipulation, we must set the `noCursor` field of the control to 1, or true.

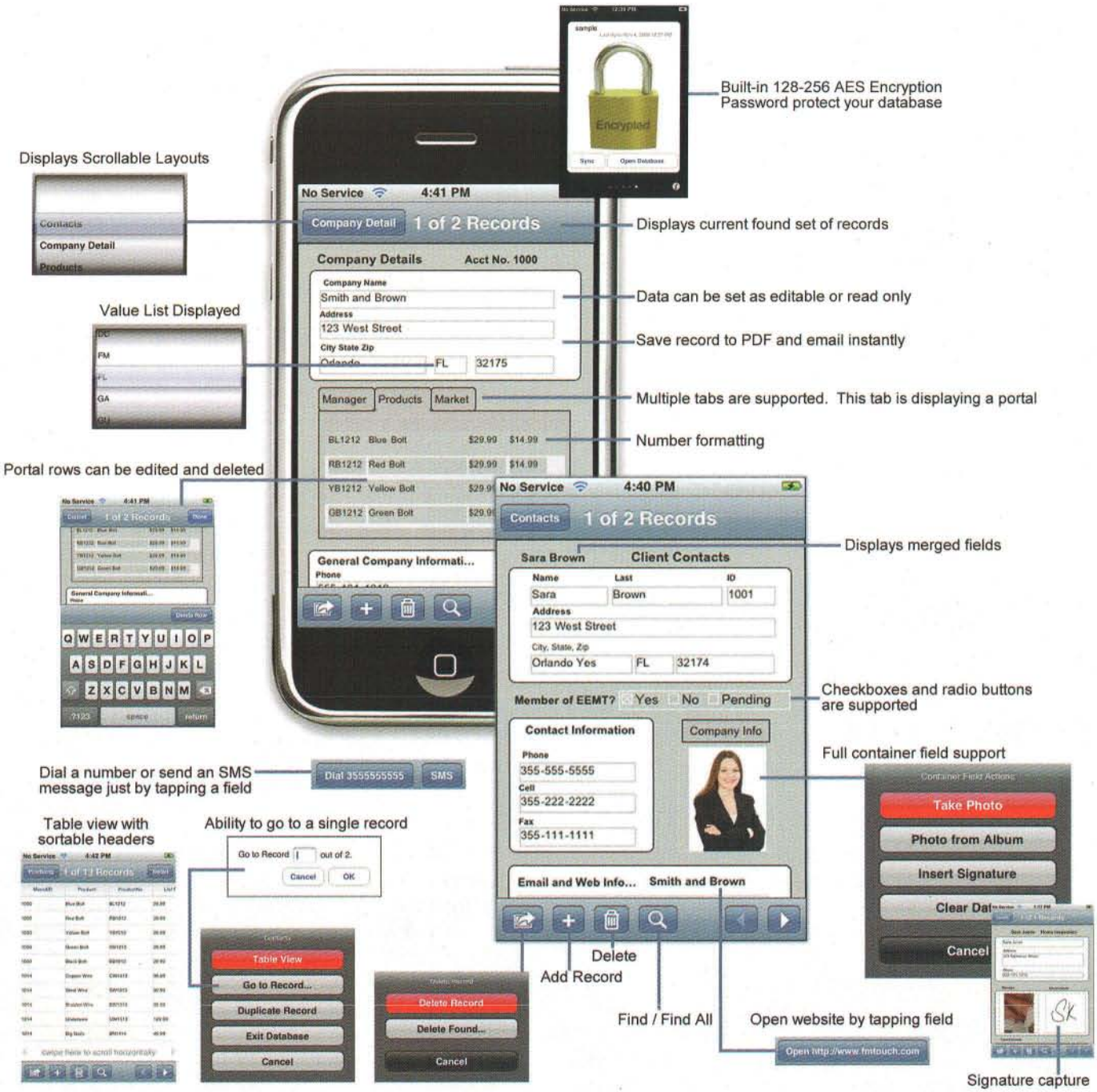
Then we define a method for the `PlayerInterface` control that describes what to do when the control becomes active ("wakes up"). It's not much, but what it does is activate DirectInput in order to grab any user inputs at the keyboard or mouse and then make the `playerKeymap` bindings active. The profile called GuiContentProfile is predefined for us in the common code base. Of course we would be free to make our own profile if we want. DirectInput? On a Mac?! Fear not! Torque uses a DirectX wrapper to handle all the stuff for that Other System. It does the appropriate thing on a Macintosh.

Next, we define a callback method for the `GameConnection` object (you know, the one we created back there in `game/main.cs`). The engine invokes this method internally when the server has established the connection and is ready to hand control over to us. In this

FMTouch

It's like having FileMaker Pro® in your pocket

For the first time ever, you can have the full relational power of FileMaker Pro in your pocket. With FMTouch you always have your data with you. So no matter who you are or what you do, you can be assured that the information that you need is always just a touch away.



FMTouch enables you to harness the power of FileMaker Pro® on your Apple iPhone® or iPod Touch®
FMTouch is available at the Apple Store More information is available at www.fmtouch.com

method we assign our player interface control to the Canvas we created earlier in the `InitializeClient` function in the `game/initialize.cs` module.

After that, we define a whole raft of motion functions to which we will later bind keys. Notice that they employ global variables, such as `$mvLeftAction`. This variable and others like it, each of which starts with `$mv`, are seen and used internally by the engine.

Then there is a list of key bindings. Notice that there are several variations of the `Bind` calls. First, there are binds to our `playerKeymap`, which makes sense. Then there are binds to the `GlobalActionMap`; these bindings are available at all times when the program is running, not just when an actual game simulation is under way, which is the case with a normal action map.

When a function mapped to a `Bind` is called, the function is invoked with a single argument. If the key was just pressed, then the argument is set to `true`. If the key has just been released, the argument is set to `false`.

The `BindCmd` method is a little different. It accepts *two* callback arguments after the key specifier. The first argument contains a callback function to be executed when the mouse was pressed down. The second argument contains the callback function to be executed when the key is released.

Finally, there is a list of stub routines. All of these routines are called from within the common code package. We don't need them to do anything yet, but as before, in order to minimize log file warnings, we create stub routines for the functions.

MacinTorquin'

Once you've typed in all the modules, you should be in a good position to test MacinTorque. The `game` package in MacinTorque is a pretty minimalist package. When you launch `Torque Demo OSX`, you will be deposited directly into the game. Once you have been deposited dropped into the game, you have a small set of keyboard commands available to control your avatar, as shown here:

Key	Description
w	Run forward.
s	Run backward.
a	Run (strafe) left.
d	Run (strafe) right.
Spacebar	Jump.
Tab	First/Third-Person View Toggle
Z	Character-centric Free-look
Escape	Quit game.
Tilde	Open console.
Mouse	Free-look, turn character

After you have created all of the modules, you can run MacinTorque simply by double-clicking `Torque Demo OSX`. You will "spawn" into the game world above the ground and then drop down. When you hit the ground, your view will shake from the impact. If you turn your player around, using the mouse, you will see the view shown in Figure XXcountryside in all of its naked and natural glory.

After spawning, you can run around the countryside, admire the countryside, and jump.

If you hit the Tab key, you will see your character, Stick Norris, in third-person view mode.

Hold down the Z key while moving the mouse around to view Stick from different angles.

There you go

You should have a fairly simple game now. I'll be the first to admit that there is not much to do within the game, but then that wasn't the point, really. By stripping down to a bare-bones code set, we get a clearer picture of what takes place in our script modules.

By typing in the code presented in this article, you should have added the following files in your MacinTorque MACINTORQUE folder:

```
/main.cs
/game/main.cs
/game/client.cs
/game/server.cs
/game/initialize.cs
/game/playeranimations.cs
```

The program you have will serve as a fine skeleton program upon which you can build *your* game in the manner that *you* want.

By creating it, you've seen how the responsibilities of the client and server portions of the game are divvied out.

You've also learned that your player's avatar needs to have a programmatic representation in the game that describes the characteristics of the avatar and how it does things.

In the next article we will expand this simple game by adding game play code on both the client and the server sides.

MT

About The Author

Ken Finney is a GarageGames Associate Developer, co-founder of TubettiWorld Games Inc., author of* 3D Game Programming All In One *and other books about creating games, and teaches Game Art, Design, Development and Programming at the University of Ontario Institute of Technology, and the Art Institute of Toronto, in Ontario, Canada. Ken spent 20 years creating advanced technology in the industrial and commercial software engineering world before getting involved in game development full-time. You can reach Ken at ken.finney@tubettiworld.com. ***TubettiWorld Games have two game projects in the pipeline,* Return to TubettiWorld *and* Juggernaut.**

Brush Your Dog's Teeth Lately?

Replaced your smoke alarm batteries? Copied your files to the server? Laptop backed up?

Despite the best of intentions, there are some things we just don't do often enough. And when comes to backup, your business continuity is at risk.

CrashPlan PRO is the first and only backup solution that combines an extremely people-friendly client with a sophisticated enterprise server to continuously protect your business onsite, offsite, and online.

Mission critical data on remote laptops, desktops and servers are backed up in real-time to multiple destinations regardless of location.

Try CrashPlan PRO in a free 30-day trial and make life easier for you, your users, and their dog's teeth.

People Friendly. Enterprise Tough.
www.crashplanpro.com

CRASHPLANPRO™
Continuous Backup for Business.

by Dave Dribin

Quit Bugging Me

Debugging on Mac OS X

The Perfect Story

You've been a faithful reader of *The Road to Code*. You've typed in every code example to gain complete understanding. If you had problems getting something working, you could always download the source code from the MacTech web site. But now, you're off on your own. You've started writing your own applications, and inevitably, something goes wrong. Your application doesn't work as you expect it. Or it crashes, losing all your hard-entered data. How do you figure out what's going wrong?

The process of fixing an application that does not work correctly is called *debugging*. The term debugging in the context of computer software is often credited to an early computer scientist named Grace Hopper. As the legend goes, she was working on the Mark II system at Harvard, investigating a glitch in the system. The problem ended up being a moth stuck in one of the relays, and she dubbed this the "first actual case of bug being found". From then on, whenever a computer system has not worked, it is blamed on bugs. Thus, removing bugs in a computer program is called debugging.

That's a fine and dandy story, but how do you actually debug a program? It's not as if you will usually find real moths stuck inside your computer to eradicate. Let's go over some techniques that you can use to help you debug your own code.

Simple Logging

One of the simplest debugging techniques is to use *logging*. Say some variable is not being set correctly, and you do not know why. You can begin tracking down the problem by inserting `NSLog` statements to print its value out to the console. You can even use `NSLog` in a GUI application and then view the output in the Xcode console. This kind of debugging, while primitive, is surprisingly effective. It's also one of the easiest to implement, since printing to the console is one of the first things you learn how to do.

Sometimes this kind of debugging is called `printf` debugging, because in straight C you use the `printf` function instead of `NSLog`. Using `NSLog` in Objective-C has some nice advantages, but the main one is the `%@` format pattern to print objects. You can use the `%@` format string to print any Objective-C object. For example, to print an `NSString` variable:

```
NSString * name = @"Joe";
NSLog(@"name: %@", name);
```

You've seen code like this before. The `%@` in the format string gets substituted with the string value:

```
2008-12-07 22:49:09.356 Rectangles[60874:10b] name: Joe
```

Along with the message you pass as its arguments, `NSLog` prints the application name and a timestamp. We've seen how you can use other format patterns, such as `%d` to print integers and `%f` to print floating point numbers. The `%@` formatter works on any class, not just `NSString` though. Here's how you could print an `NSNumber` instance:

```
NSNumber * number = [NSNumber numberWithInt:42];
NSLog(@"number: %@", number);
```

The output would look like this:

```
number: 42
```

I'm omitting the timestamp and application name from now on, to keep the output succinct, but there's nothing too surprising going on here. You get useful output on a surprising number of classes included in Foundation and AppKit. For example, logging an `NSArray` instance displays the contents of the array:

```
NSArray * array =
   [NSArray arrayWithObjects:@"Jack", @"Jane", nil];
NSLog(@"array: %@", array);
```

The output would look like this:

```
array: (
    Jack,
    Jane
)
```

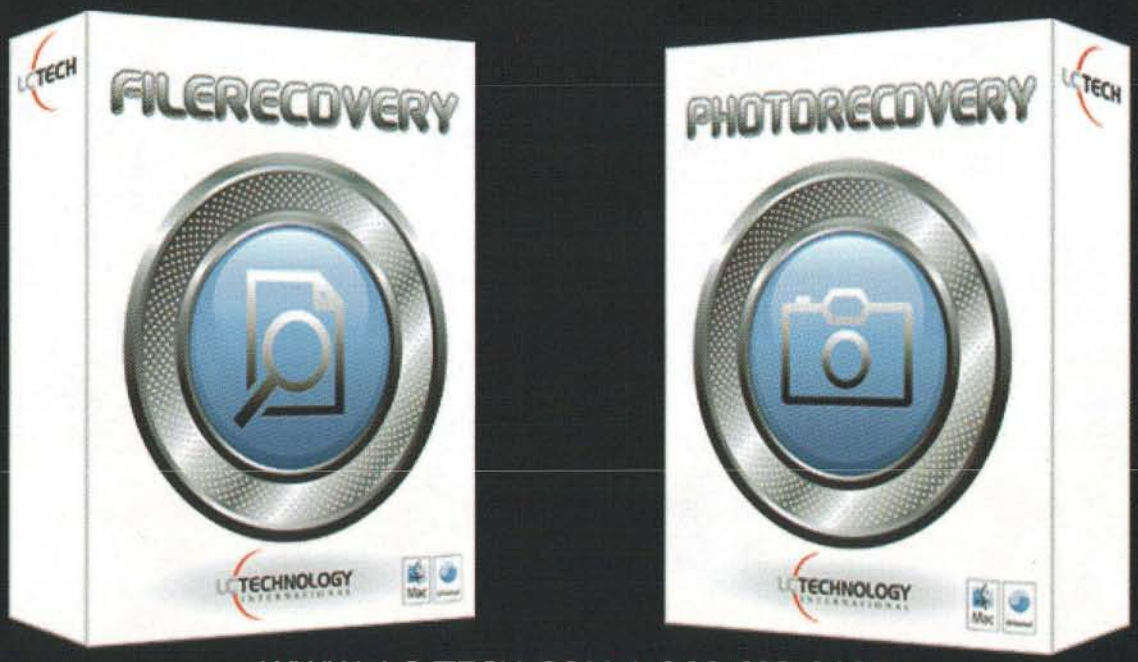
You can even use `%@` on your own objects; however, it doesn't always provide useful information. Let's use the Rectangles application from the October 2008 issue, as an example. You can get it from the MacTech FTP site, if you need to. Here's how you could log one of our `Rectangle` objects:

```
Rectangle * rectangle =
    [[Rectangle alloc]   initWithLeftX:0
                                bottomY:0
                                rightX:15
                                  topY:10];
NSLog(@"rectangle: %@", rectangle);
```

And here's what it looks like by default:

```
rectangle: <Rectangle: 0x104fba0>
```

DATA RECOVERY FOR YOUR MAC



WWW. LC-TECH.COM 1-866-603-2195

LC TECHNOLOGY INTERNATIONAL

Mac

Universal

Hrm... not very useful. It's printing out the pointer value of the object. Luckily, we can customize this output. In order to turn an object into a string, NSLog uses the description method defined on NSObject. The default implementation prints the class name and the pointer value of self, and it would look something like this:

```
- (NSString *)description
{
    NSString * description =
        [NSString stringWithFormat:@"<%@: %p>", [self
className], self];
    return description;
}
```

It's using the className method, also defined on NSObject, to get the name of the class as a string. The %p format specifier prints out pointer values. The output is the hexadecimal number of the memory address of the pointer.

By overriding description in our Rectangle class, we can give it more meaningful output. As a refresher, Listing 1 shows the interface for the Rectangle class.

Listing 1: Rectangle.h header file

```
#import <Foundation/Foundation.h>

@interface Rectangle : NSObject <NSCoding>
{
    float _leftX;
    float _bottomY;
    float _width;
    float _height;
}
@property float leftX;
@property float bottomY;
@property float width;
@property float height;
@property (readonly) float area;
@property (readonly) float perimeter;

- (id) initWithLeftX:(float)leftX
             bottomY:(float)bottomY
              rightX:(float)rightX
                topY:(float)topY;

@end
```

A useful implementation of description would output the leftX, bottomY, width, and height:

```
- (NSString *)description
{
    NSString * description =
        [NSString stringWithFormat:
         @"<%@: (%.1f, %.1f) %.1f x %.1f>",
         [self className],
         _leftX, _bottomY, _width, _height];
    return description;
}
```

Now, when we log our rectangle variable above, it would look like this:

```
rectangle: <Rectangle: (0.0, 0.0) 15.0 x 10.0>
```

File Transfer & Management

Simple, Secure, & Sensibly Priced

Version 2.0 Out Now

FileGenius expertly hosts and manages your own private file transfer and job management website. Simply copy your job or customer files to the site and they're instantly – and securely – available from any web browser. It's so simple your entire organization can be up and running in minutes – with no training, and without the headaches and hassles of email and FTP.

Send large files of any kind, up to 2GB, easily and securely without firewall, bandwidth, or email limitations.

Mac and Windows compatible — only a web browser is needed.

White-labeled to promote your business, not ours.

Unlimited private workspaces can be created for any number of users or groups like clients, vendors and associates.

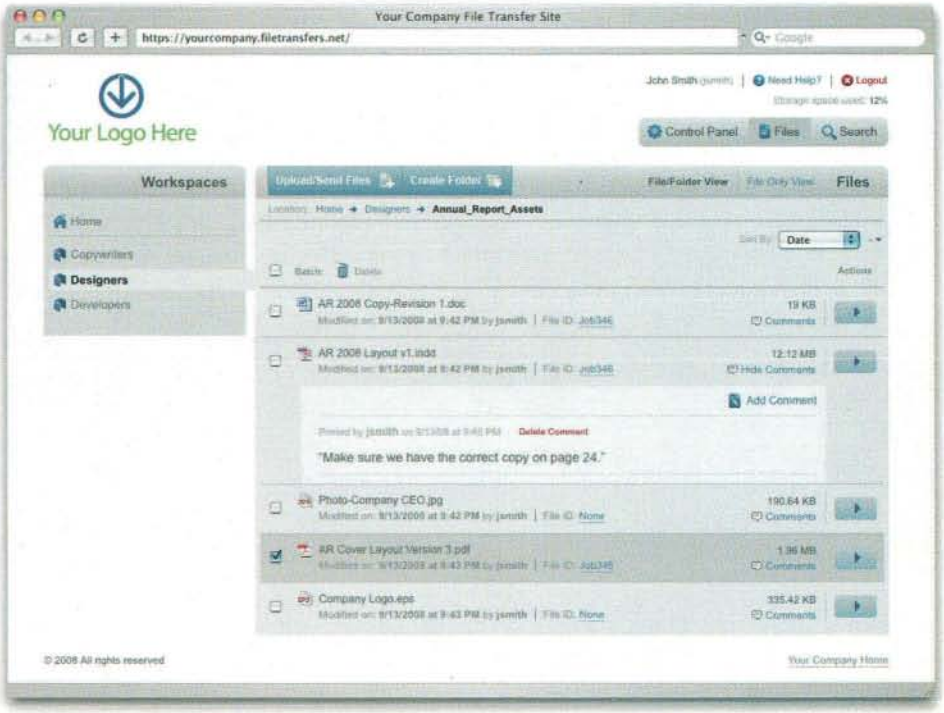
Professional, graphical interface that's as intuitive as your Mac.

So easy to set-up & use, there's no need for ongoing IT support and never any management or support costs.

Business-only file management, scalable to any number of users, perfect for both small and large organizations.

All transactions encrypted with 128/256-bit SSL encryption, plus many more security features.

The FileGenius interface.



Check out FileGenius firsthand. Try the free, no commitment 2-week TRIAL of a fully hosted, live site (ready in minutes). NO credit card information is required and NO sales personnel will call.

www.filegenius.com/mactech

FileGenius®

Ah! Much better. We can now use logging to dump our rectangles out to the console.

There are, of course, downsides to using logging for debugging. If you need to view a large number of variables or watch a variable change from line to line, you'll need to insert lots of logging statements. This is tedious and takes time. You also may accidentally leave your debug logging statements inside a shipping application. This can clutter up the user's console and even affect the performance of your application, so it is not a good idea to have `NSLog` statements in a released application. Don't fret. There's something better called a *debugger*.

Xcode Debugger

A debugger is an application whose purpose is to help debug another application. The main feature of a debugger is the ability to set *breakpoints*. Breakpoints allow you to stop the execution of a running application and investigate the state of the application. You can typically examine any local variables, instance variables, and global variables from within a debugger.

Xcode comes with an integrated debugger, meaning that you can use Xcode to set breakpoints. The simplest way to explain how it works is to provide an example. Take this code that is the constructor for the `Rectangle` class:

```
- (id) initWithLeftX:(float)leftX
             bottomY:(float)bottomY
              rightX:(float)rightX
                topY:(float)topY
{
    self = [super init];
    if (self == nil)
        return nil;

    _leftX = leftX;
    _bottomY = bottomY;
    _width = rightX - leftX;
    _height = topY - bottomY;

    return self;
}
```

Say we want to see if the rectangle is created properly. You simply click on the left margin with the line numbers, and a blue arrow is added to indicate that you added a breakpoint, as in Figure 1.

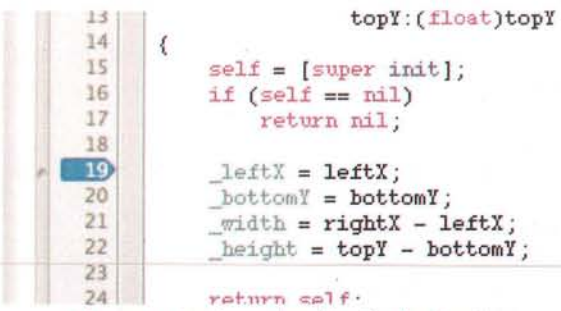


Figure 1: Setting a breakpoint in Xcode

To trigger the breakpoint, we first need to run our application with the debugger instead of running it normally. You do this by choosing the **Build > Build and Debug** menu, or **Command-Y**, instead of the usual **Build > Build and Run** menu, or **Command-R**. This will launch your application, but whenever code that has a breakpoint set is executed, the program stops execution. In order to trigger our example breakpoint, you need to hit the **Add** button in the user interface, which ends up creating a new rectangle.

When you do this, the debugger will stop your program, and Xcode will change the main editor interface to show you that you have stopped at a breakpoint, as shown in Figure 2. You can see that it adds a red arrow in the left margin and it highlights the line that the red arrow is on in blue. This is done to clearly indicate where the debugger has halted your application.Debugger integration within the standard editing window is new to Xcode 3. There is also a separate debugging window, which contains more debugging information. You can view this window by choosing the **Run > Debugger** menu, which looks like Figure 3.

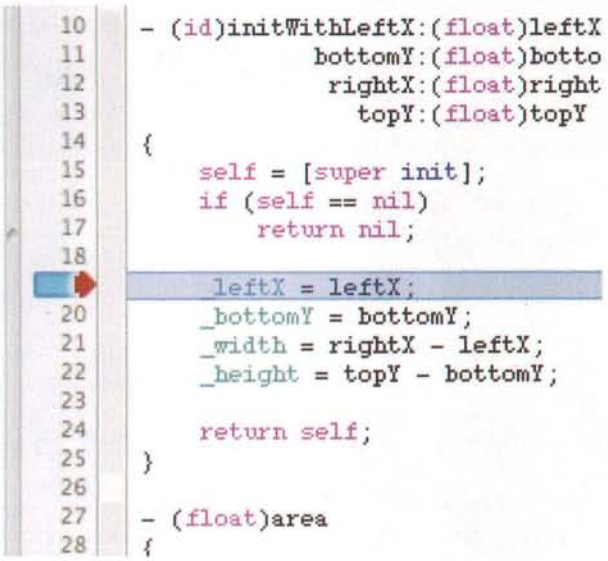


Figure 2: Xcode stopped at a breakpoint

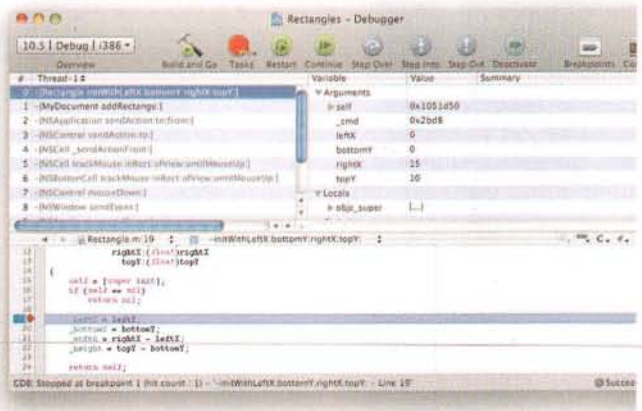


Figure 3: Xcode debugger window

BookEndz® Docking Stations for Apple Computers

DOCKING STATIONS FOR APPLE COMPUTERS

Convert your MacBook Pro®, MacBook®, or PowerBook®
into a desktop in seconds
without misplacing cables or damaging connectors.

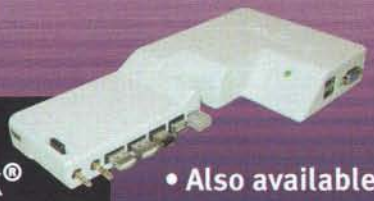
- Aluminum Plate helps in cooling of notebook
- Connectors are routed to rear of Dock
- MagSafe® Housing prevents accidental disconnect

17" MacBook® Pro Docking Station

17" PowerBook® G4 Docking Station

15.4" MacBook® Pro Docking Station

15" PowerBook® G4 Docking Station



13.3" MacBook® Docking Station

- Also available in Black
- 5 USB 2.0 compliant ports
  (4 port powered or unpowered hub, AC/DC adapter included)
- Built in DVI/VGA full size
- Gigabit Ethernet RJ45

Visit our website for latest product announcement www.BookEndzdocks.com

BookEndz®
Manufactured by Olympic Controls
1250 Crispin Drive • Elgin, Illinois 60123 • USA
Phone: 847-742-3566 • Fax: 847-742-5686 • Toll Free: 888-622-1199
E-mail: Sales@BookEndzdocks.com

While your program is halted, you can poke around and investigate its state. The lower pane of the debugger window is similar to the main editor window in that it shows you your code, along with the red arrow and blue highlight to indicate where you application stopped. The upper left pane of the debugger window shows the method call stack, or *stack trace*. This shows all method calls that lead you to this point, all the way back to `main` function that started the application.

The upper right pane of the debugger window allows you to examine the value of any variable. Variables that contain other variables, such as structures or objects, have a little disclosure triangle. Clicking on the triangle displays the containing variables. For example, clicking on the triangle of the `self` variable allows us to examine the instance variables, as shown in Figure 4. As you can see, all instance variables are set to zero initially.

Variable	Value	Summary
▼ Arguments		
▼ self	0x1051d50	
▶ NSObject	{...}	
_leftX	0	
_bottomY	0	
_width	0	
_height	0	
_cmd	0x2bd8	
leftX	0	

Figure 4: Examining variables

Examining variables is extremely helpful when trying to figure out how your application is misbehaving, but there is a lot more you can do with a debugger. Most likely, you are going to want to start your application up again. There are two ways to resume execution of your program: continuing and stepping. Continuing starts the application back up again, and it will resume as if nothing happened.

Stepping allows you to continue execution temporarily and execute the statements of your program one at a time. By stepping through your program, you can watch how each statement modifies variables. Say we step over the four statements that assign the instance variables. The debugger window would now look like Figure 5.

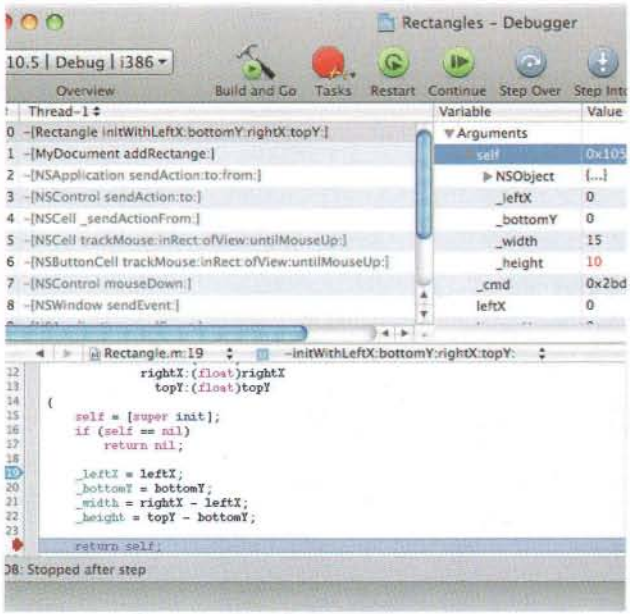


Figure 5: Stepping over statements

You will notice that the red arrow and blue highlight move as each statement is executed. Also the variable examination pane highlights variables that have changed in red. Thus, the `_height` instance variable is red because we just assigned to it. You can keep stepping as long as you would like and watch your application execute in slow motion.

There are actually two kinds of stepping: *step into* and *step over*. Both continue execution temporarily, but the difference is how the debugger handles method and function calls. Using "step into" the debugger will follow execution into method and function calls and stop at the first statement inside the called method. "Step over" will not follow execution into method and functions calls, and instead will stop on the next line of the current method. If the statement is not a method or function call, then "step into" and "step over" are equivalent.

The toolbar of the debugger window has buttons labeled **Continue**, **Step Over**, and **Step Into** so you can easily restart your application using one of these commands. It's worth noting that you can only step into your own code. Any code that is part of the system frameworks cannot be stepped into.

The upper left pane that shows the method call stack allows you to "back up" the method call chain and view the state of each method, too. By default, the top of the stack is selected. For example, if I click on the second method in the stack, `-[MyDocument addRectangle:]`, the debug window would change to look like Figure 6. Again, you can only select method calls that are in your code. You cannot select methods that are part of the system frameworks, and they are grayed out.

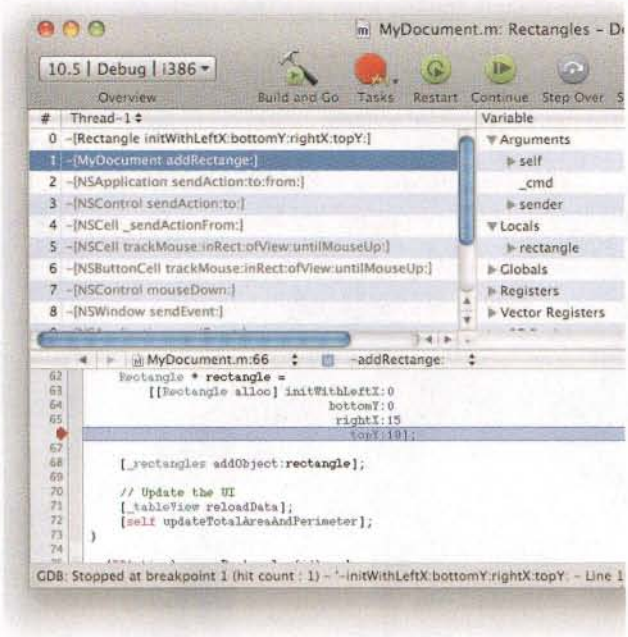


Figure 6: Selected methods in the call stack

The source pane and the variable pane update to the new method when you click on one. Thus, the variable arguments and locals shown in Figure 6 are for the `addRectangle:` method.

Debugger Console

There's another important part of Xcode's debugger called the *debugger console*. This is the same console where logging output usually goes, but with the debugger active, it's turned into a command line interface for the debugger.

Xcode's debugger is built around a debugger called *gdb*, the GNU debugger. gdb is traditionally run from the command line to debug C and C++ programs. Xcode wraps most of gdb's functionality with an easy to use GUI, but the console allows you to enter commands directly to gdb. The `print` command, for example, prints the value of a variable. You can use this instead of the variable pane in the GUI to examine variables. You can examine the `rightX` local variable from the console like this:

```
(gdb) print rightX
$4 = 15
```

The `print` command only works on primitive types, though. If you want to examine Objective-C objects, you need to use the `print-object` command. It uses the same `description` method that `NSLog` uses to get the string value of an object, so if we used this on `self` at the end of the `Rectangle` constructor, the output would look like this:

```
(gdb) print-object self
<Rectangle: (0.0, 0.0) 15.0 x 10.0>
```

You can also use `po` as a shortcut for `print-object`:

```
(gdb) po self
<Rectangle: (0.0, 0.0) 15.0 x 10.0>
```

The arguments to the `print` and `po` commands are not limited to printing variables. You can traverse structures and even call Objective-C methods. To call methods, you use the standard square bracket syntax you normally use. Here's how you would print the class name of `self`:

```
(gdb) po [self className]
Rectangle
```

Note that the debugger does not understand property dot notation. You have to use the square bracket syntax:

```
(gdb) print self.area
There is no member named area.
(gdb) print [self area]
$6 = 150
```

Generally `print` is smart enough to figure out the type to print out, but sometimes you need to give it a hint. If you need to do this, you can use standard C cast syntax to force the value to be a certain type:

```
(gdb) print (int)[self area]
$7 = 150
```

The command line is powerful, yet complex. Everything available in the Xcode debugger GUI is available from the gdb console. For example, the commands that correspond to the Continue, Step Over, and Step Into of the GUI are `continue`, `next`, and `step`, respectively.

Breakpoint Actions

One use for gdb commands is to assign actions to breakpoints. Instead of breakpoints stopping the program and not continuing right away, you can add actions to breakpoints. These actions are executed automatically when the breakpoint is triggered. Actions can be gdb commands, log statements, or even AppleScripts.

Xcode has some pre-configured breakpoint actions that you can use when setting a breakpoint. The easiest way to access these is to right click (or Control click) in the left margin of the source editor, and choose Built-in Breakpoints from the contextual menu, as shown in Figure 7.

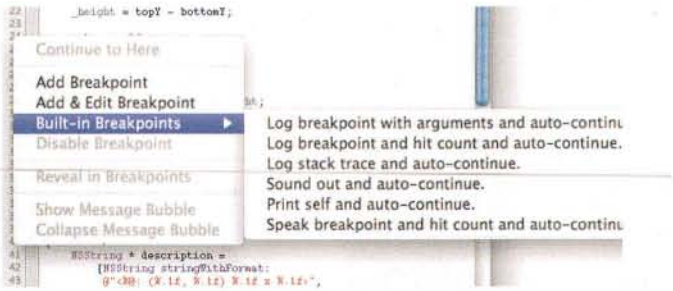


Figure 7: Built-in Breakpoints

Premium Small Business Management and Accounting Software

...and now

Point of Sale for Mac
MYOB Checkout

MYOB

Mind Your Own Business. Smarter.

800 322 MYOB (6962)
www.myob-us.com

Let's choose **Print self and auto-continue** to see how it works. Now, when we run our program and cause the breakpoint to trigger, it will log the description of the `Rectangle` and continue. This is just like adding an `NSLog` statement, except you do not have to modify your code!

Let's see how this works behind the scenes. Right click on the blue breakpoint arrow in the left margin and choose **Edit Breakpoint** from the contextual menu. This will bring up the **Breakpoints** window as shown in Figure 8. Alternatively, you can access this window from the **Breakpoints** toolbar item or the **Run > Show > Breakpoints** menu.

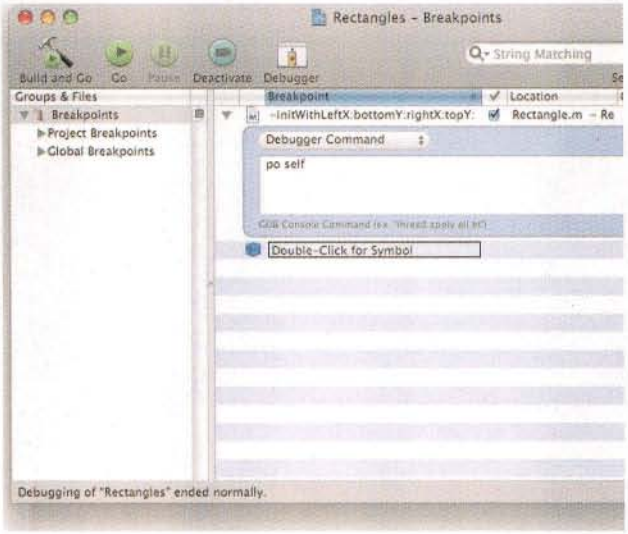


Figure 8: Breakpoint with an action

This window shows that the `po self` debugger command is automatically executed when the breakpoint is hit. Also, the rightmost column is checked, meaning the application is automatically continued. This combination is how you can add logging statements without modifying the code. Again, you can use any gdb command you can type at the console as an action, so breakpoint actions are very powerful.

Debugging Crashes

While breakpoints are a great way to track down bugs, sometimes your application will unexpectedly crash, and you don't know where to start looking. The debugger can help in these cases, too. Crashes can happen for many reasons, but one of the most common is dereferencing a `NULL` pointer. In order to demonstrate this point, I'm going to make the program crash by inserting a bug.

```
- (IBAction)addRectangle:(id)sender
{
    // Dereference a NULL pointer
    int * pointer = NULL;
    *pointer = 0;

    Rectangle * rectangle =
        [[Rectangle alloc]    initWithLeftX:0
                                    bottomY:0
                                     rightX:15
                                       topY:10];

    [_rectangles addObject:rectangle];

    // Update the UI
    [_tableView reloadData];
    [self updateTotalAreaAndPerimeter];
}
```

The second line in the method will cause the application to crash when executed. To trigger this bug, all you have to do is click on the **Add** button in the GUI. If you try this, you'll notice that Xcode will catch the crash and automatically attach the debugger. The debugger will then highlight the exact line where the crash occurred, with the red arrow and blue highlight, as shown in Figure 9. You can now examine the state of your application to figure out why the crash occurred. In this case, it's obvious: `pointer` is `0x0`, or `NULL`. However, it may not be obvious in a real crash. The debugger gives you a chance to investigate what caused the crash.

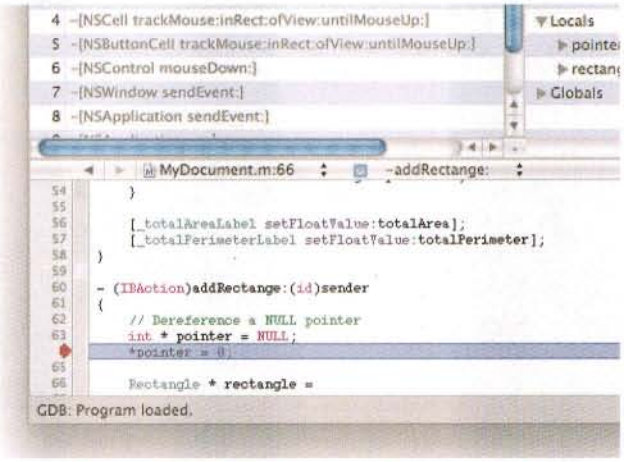


Figure 9: Xcode debugger catching a crash

The debugger will only get attached to a crashing program when running within Xcode. If a program crashes outside of Xcode, you will get a standard dialog, as shown in Figure 10.

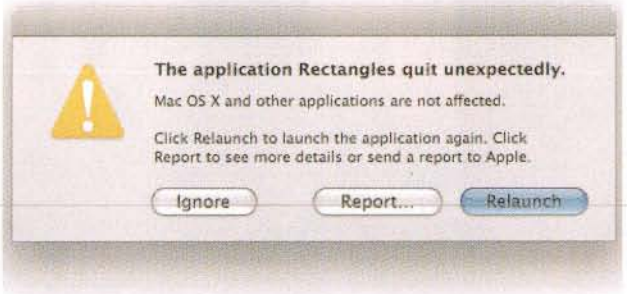


Figure 10: Application crash dialog

Mac Security Expertise Has a Name

Intego's Mac Security Solutions	
NetBarrier X5	Firewall, antivandal and privacy protection
VirusBarrier X5	Desktop antivirus protection
VirusBarrier Server	Antivirus protection for Mac OS X Server
VirusBarrier Mail Gateway	SMTP antivirus protection for Mac OS X Server
Remote Management Console	Remote management of Intego software
Personal Backup X5	Local and network backup solution
FileGuard X5	Protection for sensitive files
Personal Antispam X5	Antispam and anti-phishing protection
ContentBarrier	Lets children use the Internet safely

Intego is the Mac Security Specialist

Intego has a full line of Mac security software designed to protect Macs from the dangers of the Internet. From virus protection to firewalls, from backups to data protection, Intego is the only company specializing in keeping Macs secure. With products designed for the enterprise, only Intego can offer the kind of security that today's businesses need.

www.intego.com

There's not much information here to help you debug. However, ever time any application crashes, Mac OS X saves what's called a *crash log*. Crash logs contain information that may help determine why an application crashed, and they are stored in this directory:

```
~/Library/Logs/CrashReporter/
```

There's one text file per crash, and if you open up one of these files, you'll see the full crash log. The most useful part of the crash log is what's known as a *stack trace*. This is similar to the upper left pane of the debugger, in that it shows you the methods called in the current stack before the crash occured. The first line of the crash log (which I've edited for brevity) will look something like this:

```
Thread 0 Crashed:
0   [...] -[MyDocument addRectange:] + 29
(MyDocument.m:64)
1   [...] -[NSApplication sendAction:to:from:] + 112
[...]
11 [...] 0x00002679 main + 30 (main.m:14)
12 [...] start + 54
```

It tells us exactly which line our program crashed on. While your program has already exited, and you can't examine it for more information, the crash log is often very helpful in determining why your application crashed for a user.

You can actually configure the crash window to give you crash logs, as well, using the CrashReporterPrefs application, found in this directory:

```
/Developer/Applications/Utilities/
```

CrashReporterPrefs allows you to configure Crash Reporter for Developer mode. When in Developer mode, you can view crash logs right in the GUI when the application crashes, as shown in Figure 11. However, you get this crash dialog whenever *any* application crashes, not just ones you've written, so you may find this behavior too annoying to leave Developer mode on by default.

Figure 11: Crash dialog in Developer mode

Xcode Configurations

Take a look in the Xcode toolbar, and you'll notice the Overview item in the toolbar on the left hand side. It should say something like 10.5 | Debug | i386. To know what this means, click on the toolbar item, and you'll see a menu similar to Figure 12. You'll see that 10.5 corresponds to the Active SDK, Debug corresponds to the Active Configuration, and i386 corresponds to the Active Architecture. A full description of all these items is beyond the scope of this article, but we're going to briefly talk about the Active Configuration and Active Architecture settings.

Figure 12: Xcode overview menu

By default, Xcode uses the Debug configuration. You'll notice there is also a Release configuration. This is because your application needs to be compiled in a special manner to work properly with the debugger. When you compile with the Release configuration, your application will be faster and use less disk

space, so you will want to use it before distributing your application to your users. However, it may not work well with the debugger. You may have trouble setting breakpoints, and the stack traces included when your application crashes may not be as accurate. For example, it may not be able to tell you exactly which line your application crashed on.

Also, in the **Debug** configuration, your application is only compiled for the **Active Architecture**, which speeds up compile times. In contrast, the **Release** configuration will compile your application as a Universal binary and will run natively on both PowerPC and Intel Macs. However, this means your code is compiled twice and can slow down development time.

The **Active Architecture** is the architecture of your current hardware: if you are running on an Intel Mac, the architecture will be set to **i386**. If you're on a PowerPC Mac, the architecture will be set to **ppc**. This means that your application will only run natively on the same type of Mac that you have.

Because of the differences between **Debug** and **Release** configurations, you will typically use **Debug** while developing your application and **Release** to build the final version you ship to your users. Just be aware that you will not be able to accurately debug that final version or get accurate stack traces if your application crashes.

Conclusion

This article has given you a brief overview of how to use logging and Xcode's integrated debugger to help you diagnose and fix errors in your programs. Debugging is an art in and of itself, and just like programming, the more you do it, the better you get at it. There are lots of resources on the Internet about debugging in general as well as specifics on how to use gdb and the Xcode debugger. For instance, Apple has a good article called "Technical Note TN2124: Mac OS X Debugging Magic" on their website. This article is chock full of tips and tricks of debugging techniques for Mac OS X and a worthy read.

Bibliography

"Technical Note TN2124"
http://developer.apple.com/technotes/tn2004/tn2124.html

About The Author

Dave Dribin has been writing professional software for over eleven years. After five years programming embedded C in the telecom industry and a brief stint riding the Internet bubble, he decided to venture out on his own. Since 2001, he has been providing independent consulting services, and in 2006, he founded Bit Maki, Inc. Find out more at <http://www.bitmaki.com/> ***and*** <http://www.dribin.org/dave/>.

iPhone Productivity Applications, Part II

Developing applications that manage complex data

by Rich Warren

The Project So Far...

In the last article, we began building the GasTracker application for the iPhone. This application let us record the amount of gas, cost and odometer reading each time we fill up our car. It then calculates and display useful statistics, like average MPG or average cost per day.

We started by building the application's skeleton, the tab view and the model. By the end of the article, the application should have compiled without any errors or warnings. When you run the app, you can switch from tab to tab or customize the tab bar. Of course, the views didn't do anything yet. Still, it was a good start.

In this article, we will continue to flesh out GasTracker. We will focus on setting up the navigation controller and our table views. We will also add a view for entering data, and create custom classes for each of the stats views. Once that's done, we'll have a fully functional productivity application.

Most of this article will focus on the history view. This view is based on the 1-2 punch of a table view backed buy a navigation controller. This is a common design for iPhone applications, especially productivity apps.

The navigation controller manages a stack of view controllers. You push new view controllers onto the top of the stack. You can also pop unwanted controllers off the top of the stack. The topmost controller is active, and the navigation controller displays its view. All other views on the stack remain hidden.

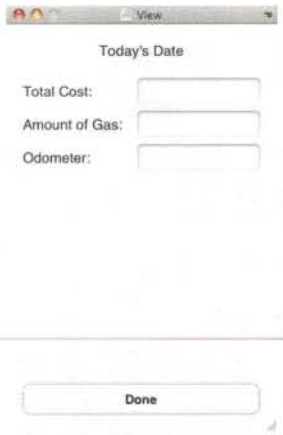
We start with a table view on the top of the stack. The table view displays all the entries we made while purchasing gas. When we tap on a row, we want to move to a detailed view for that row. We create a new view controller for the detailed view, and push it onto the stack. To get back to the main table, we just pop the controller off the stack. In this way, the table/navigation control combo let us navigate through our (albeit short) data hierarchy.

We will examine the interactions between the navigation controller and the table view in more detail later on, but before we can go any further, the user needs some way to add entries to the model.

Add Entry View

Let's start with the nib. Right click on the **Resources** folder and select **Add fi New File...**. From the **User Interfaces** templates, select **View XIB**. Name it `AddEntryView.xib` and click Finish.

Now open the nib in Interface Builder. We want to add four labels, three text fields and a single button. Organize the controls to match the image below. Drag the control from the **Library Window**, and position it on the view. As you move it around the view, notice that blue guidelines appear when you get close to the edges or to other controls, use these to help position the controls correctly.



AddEntryView.xib

To change the label text, either double click on the label and edit it directly, or select the label and change the **Text** attribute in the **Attributes Inspector**. For the **Today's Date** label, stretch the label so that it fills the view from margin to margin, then center align the text. The alignment controls can also be found on the **Attributes Inspector**.

The button's text works the same way, except it is called the **Title** attribute, and the text is already centered by default.

In general, I find it easiest to place the text fields first. Make sure they are aligned along the view's right margin with the proper vertical spacing. Then place the labels relative to the text fields. Once these are set, adjust the text field width based on the longest label, then adjust the others to match.

Interface Builder has a number of layout tools to help you. Check out Apple's **Interface Builder User's Guide**, for more information than you will probably ever need.

Now we need a controller for this view. Go back to XCode, and again right click on the **Classes** group in the **Groups & Files** tree, and select **Add fi New File...**. From the **Cocoa Touch Classes** templates, select **UIViewController** subclass. Name the class `AddEntryViewController.m`, and click on the **Finish** button.

Now, open `AddEntryViewController.h` and edit it as follows:

AddEntryViewController.h

```
#import <UIKit/UIKit.h>
@class Model;

@interface AddEntryViewController :
UIViewController<UITextFieldDelegate> {

    IBOutlet UITextField* totalCost;
    IBOutlet UITextField* amountOfGas;
    IBOutlet UITextField* odometer;
    IBOutlet UILabel* todaysDate;
    IBOutlet UIButton* doneButton;

    Model* model;

}

@property (nonatomic, assign) Model* model;

-(IBAction)done;

@end
```

Here, we define the outlets for our text fields, the **Today's Date** label and our button. We also create a property for our `Model`, and create an action for our button. Nothing too surprising.

However, before we move on to the implementation file, let's take a quick sidestep. We're going to spend a lot of time working with formatted strings. We want to use basically the same formatting and parsing methods throughout this project, in a number of different classes.

In many languages (I'm looking at you, Java), we would create a utility class to hold these common methods, but Objective-C has a better solution. We can create a category for an existing class. Categories let us add methods to existing classes. We don't even need the original class's source code.

So, let's create a category that will add our specialized formatting/parsing methods directly to NSString. Add a new file to your **Classes** group. Since there's no **Category** template, just add an `NSObject` subclass. Name the file `Formatter.m`.

Now, open `Formatter.h`. Edit it as shown below:

Formatter.h

```
#import <UIKit/UIKit.h>

@interface NSString (Formatter)

+ (NSString*)shortDate:(NSDate*)date;
+ (NSString*)longDate:(NSDate*)date;
+ (NSString*)decimal:(double)value;
+ (NSString*)currency:(double)value;
+ (double) parseDecimal:(NSString*)string;
+ (double) parseCurrency:(NSString*)string;

@end
```

This creates a category on `NSString` named `Formatter`. It then declares two methods that create formatted strings from an `NSDate` object. There are two additional methods to create formatted strings from doubles, and two methods that parse a properly formatted `NSStrings`, producing double values.

Let's define these methods in `Formatter.m`. Functionally, these are all very similar. I will show you a single formatter and its corresponding parser below. I leave the rest up to you.

Formatter.m

```
#import "Formatter.h"

@implementation NSString (Formatter)

+ (NSString*)decimal:(double)value{

    NSNumber *number = [NSNumber numberWithDouble:value];
    NSNumberFormatter *formatter = [[NSNumberFormatter
alloc] init];
    [formatter
setNumberStyle:NSNumberFormatterDecimalStyle];

    NSString *decimalString = [formatter
stringFromNumber:number];
    [formatter release];

    return decimalString;
}

+ (double) parseDecimal:(NSString*)string {
    NSNumberFormatter* formatter = [[NSNumberFormatter
alloc] init];
    [formatter
setNumberStyle:NSNumberFormatterDecimalStyle];

    NSNumber* value;
    NSString* error;

    [formatter getObjectValue: &value forString:string
        errorDescription:&error];

    [formatter release];

    return [value doubleValue];
}

@end
```

In the `decimal:` method, we create an `NSNumberFormatter`, and set its style to `NSNumberFormatterDecimalStyle`. By default, all `NSFormatter` classes set a number of attributes based on our local. In my case, it sets the decimal separator to a period ".", and the thousands separator to a comma ",". For currency formatted numbers, it would also set the currency symbol to the dollar sign "$".

While you can further modify the formatter's behavior, the default behavior will work fine for us. Simply call the `NSNumberFormatter`'s `stringFromNumber:` method, and return the resulting string.

The parser also uses `NSNumberFormatter`. As before, we create our formatter and set the desired style. Then we call `getObjectValue:forString:errorDescription:` and return the value.

Note: we completely ignore the error message here. Generally speaking, that's a bad idea. However, since we are using the same formatter to both create and parse the strings, there shouldn't be any unexpected errors. So, maybe it's ok to let it slide, just this once.

OK, back to the `AddEntryViewController`. Let's open the implementation file. First things first, let's import all the header files we'll need for this code. Also, lets define a few private methods. These are methods that our class will use internally, but that cannot (or at least, cannot without some difficulty and hacking) be called from the outside.

AddEntryViewController.m Imports and Private Methods

```
#import "AddEntryViewController.h"
#import "Formatter.h"
#import "Entry.h"
#import "Model.h"

// Define private methods
@interface AddEntryViewController()
-(double)getTotalCost;
-(double)getAmountOfGas;
-(double)getOdometer;
-(void)validateControls;
@end
```

Note: we are using an extension to define our private methods. Extensions look like categories, but without the name. They also operate similarly to categories; both let us add methods to existing classes. But, extensions work in a much more constrained way.

When you create an extension, the method definitions must appear in the class's main `@implementation` block. Essentially, they provide a compiler-checked method for declaring an API outside the main `@interface` block. They are most often used to define private methods.

Now let's look at the implementation. First we synthesize our model property. Then we define our done action. The done action creates a new entry based on the values from the view, adds the entry to the model, and then pops the `AddEntryView` from the navigation controller.

Popping a view from the navigation controller dismisses the view and returns us to the next view on the stack. Since we animated the transaction, the `AddEntryView` will slide off the right side of the screen, and the previous view will slide back in from the left.

AddEntryViewController.m Implementation

```
@implementation AddEntryViewController
@synthesize model;

#pragma mark Actions

-(IBAction)done {

    NSDate* date = [NSDate date];
    Entry* entry = [[Entry alloc] initWithTotalCost: [self
getTotalCost]
                                          amountOfGas: [self
getAmountOfGas]
                                             odometer:
[self getOdometer]

onDate: date];

    [model addEntry:entry];
    [self.navigationController
popViewControllerAnimated:YES];
    [entry release];
}
```

Now let's look at methods that override existing methods from `UIViewController`, its superclasses or the `UITextFieldDelegate` protocol. `viewDidLoad` is automatically called when a view is loaded from a nib file. At this point, the `IBOutlet` values are valid. We can therefore use this method to do additional initialization on any objects managed by the nib. In our case, we want to set `todaysDate` to a string corresponding to the current date. We then make sure the `totalCost` text field is selected, by calling `becomeFirstResponder`. This, in turn, brings up the keyboard.

viewDidLoad Method

```
#pragma mark Polymorphic Methods

- (void)viewDidLoad {
    [super viewDidLoad];

    NSDate* date = [NSDate date];
    todaysDate.text = [NSString longDate:date];

    [totalCost becomeFirstResponder];
}
```

Next we override `shouldAutorotateToInterfaceOrientation:`. Despite its long name, this is actually a simple method. If it returns `YES`, the view will automatically rotate when the user changes the iPhone's orientation. Returning `NO` prevents autorotation.

Of course, nothing is ever as simple as it first seems. Views inside a tab bar will not autorotate unless all the view controllers contained by the tab bar also return `YES`. Even a single `NO` will veto rotation for all views. Since our navigation

SecuTech
SecuTech Solution Inc.

Unikey Time

A Real Time Clock Inside

Easy Solution

On-key clock counts date and time

Driverless technology reduces customers 'support work

Both automatic (envelope without source modification) and APIs protection

Numerous sample codes in various programming languages

Greater Flexibility

Protect software, flash (swf and flv) and video files

Protection in local computers or over a network

Support all popular operating systems, including Windows, Linux, MacOS and Free BSD

Remote update and real time functionality

OEM enables flexibility of case, label, and color

A Cost-effective Choice

Competitive pricing

Experienced and efficient technical support

Life time warranty

● NORTH AMERICA: 1- 888-259-5825

● BRAZIL · EGYPT · FRANCE · GERMANY · INDIA · ITALY · JAPAN · MIDDLE EAST · SERBIA · TURKEY

SecuTech Solution Inc. | www.eSecuTech.com | info@eSecuTech.com

controller is managed by the tab bar controller, this includes any controllers pushed onto the navigation controller—even if they're not currently visible.

shouldAutorotateToInterfaceOrientation Method

```
-(BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
        return YES;
}
```

Next, we simply keep the default stub for `didReceiveMemoryWarning`. This just calls the super class. We could use this method to free up view-specific memory, but for this project we will leave the method untouched.

didReceiveMemoryWarning Method

```
- (void)didReceiveMemoryWarning {
    // Releases the view if it doesn't have a superview
    [super didReceiveMemoryWarning];
    // Release anything that's not essential, such as
cached data
}
```

Our controller also acts as a delegate for the text field. `textFieldDidEndEditing:` is automatically called whenever a text field loses first responder status. Here, we determine the numeric value for the text field, then format the number using either the currency or the decimal format, as appropriate.

Actual formatting is done using the `currency:` and `decimal:` methods defined in our `NSString` category. Note: if the user enters an invalid value, the field is set to 0.0. After formatting the text field, we validate the current value of all the fields. We'll look more closely at the `validateControls` method later.

The second delegate method, `textFieldShouldReturn:`, is called automatically whenever the user taps the return button. In our case, the text field simply gives up its first responder status. This dismisses the keyboard.

textFieldDidEndEditing: Method

```
- (void)textFieldDidEndEditing:(UITextField *)textField {

    double value = 0.0;
    NSString* text = textField.text;

    if ([text length] > 0) {
        value = [text doubleValue];
    }

    if (textField == totalCost) {
        textField.text = [NSString currency:value];
    }
    else {
        textField.text = [NSString decimal:value];
    }

    [self validateControls];
}

- (BOOL)textFieldShouldReturn:(UITextField *)textField {

    [textField resignFirstResponder];
    return NO;
}
```

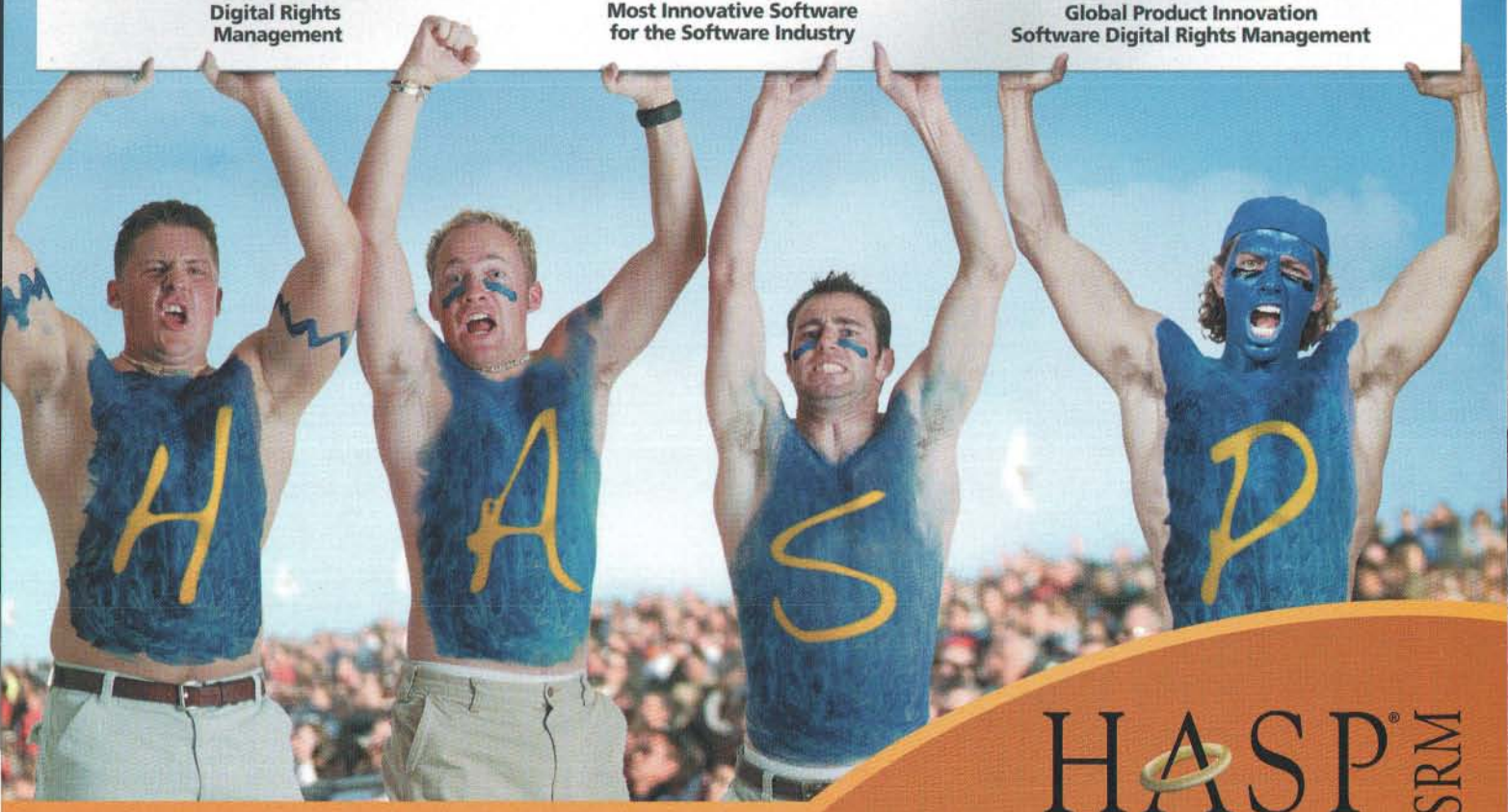
We've got some serious fans.

Get on the winning team with the industry's award-winning choice for software digital rights management.

2008 SIIA //CODiE// WINNER
Digital Rights Management

SoftwareCEO Innovation Award 2008 Winner
Most Innovative Software for the Software Industry

2007 FROST & SULLIVAN Global Product Innovation Award
Global Product Innovation Software Digital Rights Management



HASP® SRM
SOFTWARE RIGHTS MANAGEMENT

Strong software protection. Secure & flexible licensing. Trust your software security to HASP SRM: the world's #1 software protection AND licensing solution*

- Protect your software against piracy and illegal use
- Shield your valuable Intellectual Property from reverse-engineering & theft
- Create new business models with secure & flexible licensing
- Fully integrate protection & licensing with your software product lifecycle

Now Enabling Business for SaaS Vendors!

Request a FREE HASP SRM Developer Kit at www.Aladdin.com/HASPMacTech

*Frost & Sullivan N1AF-70, IDC #34452

Aladdin®
SECURING THE GLOBAL VILLAGE

NORTH AMERICA: 1-800-562-2543, 847-818-3800
UK · GERMANY · ISRAEL · BENELUX · FRANCE · SPAIN · ITALY · INDIA · CHINA · JAPAN · PORTUGAL

Aladdin.com/HASP

Finally, we have our `dealloc` and private methods. `dealloc` simply releases all the `IBOutlets`. **Note:** the memory retention rules for nib-created objects are somewhat different on the iPhone than on Mac OS X. Here, objects are created with a retain count of 1 and then autoreleased. Views retain their subviews. Additionally, when an object is assigned to an `IBOutlet`, the value is set using the `setValue:forKey:` method. This method calls the appropriate setter, if available. If no setter method can be found, it sets the variable directly, and retains the object.

We haven't defined a setter for our outlets. As described above, the objects are automatically retained, and it is our responsibility to release them when we're done. If you want more details on nib object memory management, check out **The Nib Object Life Cycle** section of Apple's **Resource Programming Guide**.

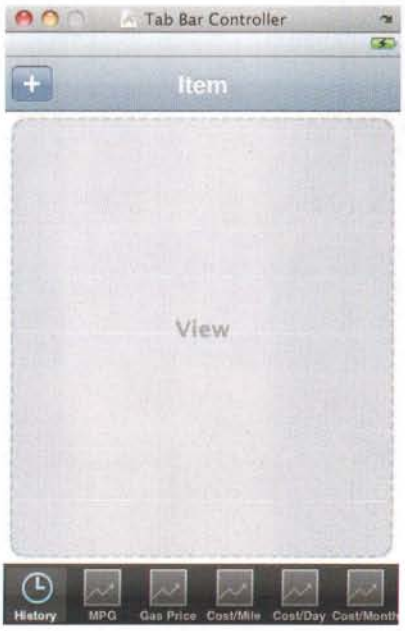
Next we have our private methods. Since we declared these methods in an extension, we must implement them in our class's main implementation block. Most of these methods simply use the currency and decimal parsing methods from our `NSString` category. `validateControls` simply verifies that all the text fields have a valid value greater than 0.0. Once all the fields have valid entries, it enables and highlights the done button.

dealloc and private methods

```
#pragma mark Dealloc

- (void)dealloc {

    [totalCost release];
    [amountOfGas release];
    [odometer release];
    [todaysDate release];
    [doneButton release];

    [super dealloc];
}


#pragma mark Private Methods

-(double)getTotalCost {

    return [NSString parseCurrency:totalCost.text];
}

-(double)getAmountOfGas {

    return [NSString parseDecimal:amountOfGas.text];
}

-(double)getOdometer {

    return [NSString parseDecimal:odometer.text];
}

-(void)validateControls {

    BOOL enable = [totalCost.text length] > 0;
    enable &= [amountOfGas.text length] > 0;
    enable &= [self getTotalCost] > 0.0;
    enable &= [self getAmountOfGas] > 0.0;
    enable &= [self getOdometer] > 0.0;

    doneButton.enabled = enable;
    doneButton.highlighted = enable;

}

@end
```

Now, go back to Interface Builder and make all the necessary connections. Make sure the **File's Owner**'s class is set to `AddEntryViewController`. Connect the **File's Owner**'s view outlet to the **view** object. Then connect `totalCost`, `amountOfGas`, `odometer`, `todaysDate` and the `done` outlets to the corresponding controls. Set the **File's Owner** as the delegate for each of the text field, and set the **File's Owner**'s `done` action to the button's `Touch Up Inside` event. Oh, one last thing. Make sure the button's **Autosizing** has a strut set on the left and bottom, and no strut set at the top. This will keep the button positioned relative to the bottom of the view, which prevents it from being hidden behind the tab bar.

So far, so good. But we need a way to launch our control. Open `MainWindow.xib` and click on the **History** tab. Drag a **Bar Button Item** to the left side of the **Navigation Bar**. In the **Attributes Inspector**, change the button's **Identifier** to **Add**. It should match the picture below.



Now add the following function to the `HistoryNavigationController`. Don't forget to declare the method.

HistoryNavigationController's addEntry Method

```
-(IBAction)addEntry {

    AddEntryViewController* subview =
        [[AddEntryViewController alloc]
```

STUFFIT® 2009

YEAR ANNIVERSARY

THE ULTIMATE IN COMPRESSION

Compress Photos, MP3s, PDFs & Microsoft Office Documents

Expand & Create 100% Compatible Windows Archives

Integrated with Quick Look & Time Machine

Learn more at StuffIt.com

smithmicro
software

```
initWithNibName:@"AddEntryView"
        bundle: nil];

    subview.model = self.model;
    [self pushViewController:subview animated: YES];

    [subview release];
}
```

This simply creates the `AddEntryViewController` using the `AddEntryView.xib` file. We set the controller's model property, and then push the controller onto our navigation controller. As described earlier, pushing a view controller onto the navigation controller makes it the current active view. Since we are animating the transition, `AddEntryView` will slide in from the right, while the main history view table slides off the left.

Now, link the button and action. You can access the `HistoryNavigationController` by single clicking on the History tab bar. If you end up with the Tab Bar Item selected, simply click another tab, then single click History again. Now right click the history tab and draw the connection between the `addEntry` action and the button.

Congratulations, you can now add new entries. You can't view them yet, but it's still progress. Next we look at the history view.

History View

We want to display all of our entries in a table view. To do this, create a `UIViewController` subclass named `HistoryViewController`. Edit `HistoryViewController.h` as shown below:

HistoryViewController.h

```
#import <UIKit/UIKit.h>
@class Model;


@interface HistoryViewController : UITableViewController
    <UITableViewDelegate, UITableViewDataSource> {

    IBOutlet Model* model;
}

@property (nonatomic, retain) Model *model;

@end
```

Note, we change the superclass to `UITableViewController`. We also adopt both the `UITableViewDelegate` and the `UITableViewDatasource` protocols. As we will soon see, these methods are used to fill, format and control our table's behavior.

Small Tree COMMUNICATIONS

Sharing has never been Easier!

SMALL TREE COMMUNICATIONS INTRODUCES:

GraniteSTOR℠ *Products*

Expand your potential with Small Tree's Innovative and Affordable products

Ethernet-based Shared Storage Solutions for professional Video and Audio Editors

- Affordable Archive storage solution using AoE
- Fast, feature rich shared storage solution using iSCSI
- Fiber Channel performance for Ethernet cost
- Watch your Final Cut Pro, Pro Tools and Photoshop applications run ***Faster*** with GraniteStor℠ products

Visit us at MacWorld in the North Hall Special Interests Pavilion, booth 3534

Small Tree - Making work more like play everyday

Small Tree COMMUNICATIONS

Small Tree Communications • 7300 Hudson Blvd., Suite 110, Oakdale, MN 55128 • 866.STC.1MAC • e-mail:childsplay@small-tree.com

Note: The view controller, table view delegate and table view data source do not have to be the same class. In some cases, you may want to separate out some of these responsibilities to other classes. However, I often find it convenient to keep them together.

OK, now lets look at the implementation:

HistoryViewController.m imports and properties

```
#import "HistoryViewController.h"
#import "EntryViewController.h"
#import "Formatter.h"
#import "Model.h"
#import "Entry.h"


@implementation HistoryViewController
@synthesize model;
```

Nothing too surprising here. We import the header files and synthesize our model. The next two methods start the real work. `viewDidLoad` adds an edit button to the right side of our navigation bar. Remember, there's not a lot of space on the navigation bar. We've already added a button to create new entries on the left side, and the title is displayed in the middle.

The table view will automatically load our data the first time the table is displayed. However, the function `viewDidAppear:` gets called each time the view becomes active. By explicitly reloading the data here, we update the table view as the application transitions from the `AddEntryView` back to the `HistoryView`.

Of course, this may not be the most efficient approach. Strictly speaking, we only need to update the new row. For our purposes, the brute force approach works well enough, but if you want a more fine grained solution, `UITableView` has methods for adding and deleting single rows. It also has methods for batching updates. Check out `insertRowsAtIndexPaths:withRowAnimation:`, `deleteRowsAtIndexPaths:withRowAnimation:`, `beginUpdates` and `endUpdates` for more information.

viewDidLoad and viewWillAppear:

```
- (void)viewDidLoad {
    [super viewDidLoad];
    self.navigationItem.rightBarButtonItem =
self.editButtonItem;
}

- (void)viewWillAppear:(BOOL)animated {
   [self.tableView reloadData];
}
```

Now we get to the heart of it. The table view automatically calls the delegate and data source methods to layout its appearance and content. Most of these methods are optional, we simply implement the ones we need. Let's start with the simplest.

numberOfSectionsInTableView: and tableView:numberOfRowsInSection:

```
#pragma mark Table Methods

- (NSInteger)numberOfSectionsInTableView:(UITableView
*)tableView {
    return 1;
}


- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section {
        return [model numberOfEntries];
}
```

These methods tell the table view that we have only a single section in our table, and that we want one row for each entry in our model.

On the iPhone, tables can only have one column; however the rows can be grouped into multiple sections. Each section can have its own header and footer. If you're using a grouped style table, you can even define custom `UIViews` (e.g. `UILabels` or `UIImageViews`) for the header and footer.

Next we return an appropriately formatted cell for each row.

tableView:cellForRowAtIndexPath:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell = [tableView
        dequeueReusableCellWithIdentifier:CellIdentifier];

    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
              initWithFrame:CGRectZero
            reuseIdentifier:CellIdentifier]
              autorelease];
    }

    int row = indexPath.row;
    Entry* entry = [model getEntryForIndex: row];

    NSString* dateString = [NSString shortDate:entry.date];
    NSString* costString = [NSString
currency:entry.totalCost];

    cell.text = [NSString stringWithFormat: @"%@: \t%@",
        dateString, costString];

    cell.accessoryType =
UITableViewCellAccessoryDetailDisclosureButton;

    return cell;
}
```

First we try to reuse our cell. For efficiency, as long as the general format of the cell remains the same, you should reuse cells. Our cells are simple. We merely change the text for each row. So, we can safely reuse our cells. If we cannot reuse the cell (for example, if it doesn't exist yet) we create a new one.

Note: the table will automatically position and size the cell; in most cases you can simply pass `CGRectZero` in for the frame. If you have more complex cells (for example, cells with

multiple subviews, each having their own autoresizing mask) you may need to use a non-zero frame size to make sure everything gets positioned properly. Otherwise, stick to `CGRectZero`.

Once we have a cell, we simply get the corresponding entry based on the row number. We then create a string from the entry's date and cost, and use that to set the cell's text property.

Finally, we add an accessory disclosure button to the row. This is a round, blue button with a white chevron displayed at the right edge of the row. The iPhone SDK provides a number of accessory buttons, each with a specific intended meaning. The accessory disclosure button should be used whenever selecting the row displays detailed information about the selected item.

We have already added an edit button. This allows the user to delete rows. We must make sure our delegate catches and handles these deletions.

tableView:commitEditingStyle:forRowAtIndexPath:

```
- (void)tableView:(UITableView *)tableView
    commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
     forRowAtIndexPath:(NSIndexPath *)indexPath {

    if (editingStyle == UITableViewCellEditingStyleDelete) {

        int row = indexPath.row;
        [model removeEntryAtIndex:row];

        [self.tableView deleteRowsAtIndexPaths:
            [NSArray arrayWithObject:indexPath]

withRowAnimation:UITableViewRowAnimationFade];
    }
}
```

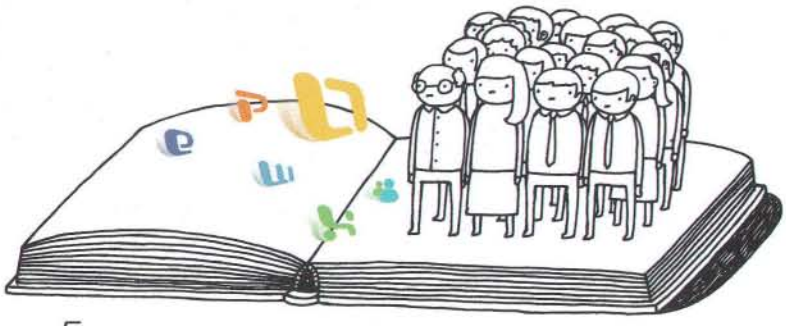
This method is called whenever the user edits the table. Here we check to make sure we're deleting a row. Then we remove the corresponding entry from our model, and delete the row with a fade animation.

We also want to display a detailed view of the entry whenever the user selects a row.

tableView:didSelectRowAtIndexPath:

```
- (void)tableView:(UITableView *)tableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath {

    int row = indexPath.row;
    Entry *entry = [model getEntryForIndex:row];

    EntryViewController *controller = [[EntryViewController
alloc]
        initWithNibName:@"EntryView" entry:entry];

    [self.navigationController pushViewController:controller
        animated:YES];

    [controller release];
}
```

Your potential. Our passion.®
Microsoft®



Everyone on the same page

Work together. Different machines? Different platforms? No matter. You can all speak the same language.

SimplifyYourWork2008.com

Microsoft® Office:mac 2008

Here, we get the corresponding entry from our model, create an `EntryViewController` using the `EntryView` nib and our entry object, then push that view onto the navigation controller. This will cause the new view to slide in from the right.

Of course, for this to work, we need an `EntryViewController` class and an `EntryView.xib` file. I will leave those as an exercise for the reader (or, if you want to cheat, you can download the complete source code from ftp.mactech.com).

OK, last and (let's be honest here) least, we have a few minor methods to round out our class.

Autorotation, memory warnings and dealloc

```
- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {

    return YES;
}


- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as
cached data
}


- (void)dealloc {
    [model release];
    [super dealloc];
}


@end
```

You should be familiar with `shouldAutorotate-ToInterfaceOrientation:` by now. Remember, when you have views inside a tab bar controller, it's an all or nothing. Unless all the views return `YES`, none of them are allowed to autorotate.

Next, we have the default stub for our memory warnings. We don't have any non-essential data that we could release, so we simply call the super class's implementation.

Our `dealloc` method simply releases our model. No surprises there.

Now we need a nib. Create a new file named `HistoryView.xib`, using the **View XIB** template. In Interface Builder, set the **File's Owner**'s class to `HistoryViewController`. Delete the **View**, and replace it with a **Table View**. Finally, draw a connection from the **File's Owner**'s view property to our **Table View** object.

Now, back in `MainWindow.xib`, single click on the history tab then single click on the view. The **Attributes Inspector** should display **View Controller Attributes**. Set the **NIB Name** to **History View**. Change the class to `HistoryViewController`. Now right click on the view, and connect the model outlet to our model object.

That's it, the history view is done. Build and run your application. You should now be able to add, view and delete entries.

Stats Views

By comparison, the various stats views are simple. We'll do one together, just so you get the idea. Let's start by adding our outlets and setters to the `StatsViewController`. Open the header file and modify it as shown below:

StatsViewController.h

```
#import <UIKit/UIKit.h>
@class Model;


@interface StatsViewController : UIViewController {
    IBOutlet UILabel* titleLabel;
    IBOutlet UILabel* valueLabel;
    IBOutlet Model *model;
}

@property (nonatomic, retain) Model *model;

        - (void)setTitle:(NSString*)title;
 - (void)setDecimalValue:(double)value;
 - (void)setCurrencyValue:(double)value;


@end
```

If this were a production application, we would probably want artistic graphs that display stock-ticker-like history lines that show our MPG and costs changing over time. While Cocoa Touch's Quartz library makes it easy to create beautiful 2D drawings, those are beyond the scope of this article. Instead, we'll simply use `UILabels` to display overall averages. The `titleLabel` contains the statistic's name, while `valueLabel` contains the average value to date. We can set these values using the `setTitle:` `setDecimalValue:` and `setCurrencyValue:` methods. Now let's look at the definitions.

viewDidLoad method

```
- (void)viewDidLoad {
    [super viewDidLoad];

    titleLabel.text = @"undefined";
    valueLabel.text = @"undefined";
}
```

This method is called after the view has loaded. It simply sets the title and value labels to "undefined".

shouldAutorotateToInterfaceOrientation: and didReceiveMemoryWarning methods

```
- (BOOL)shouldAutorotateToInterfaceOrientation:
    (UIInterfaceOrientation)interfaceOrientation {
        return YES;
}
```

```
- (void)didReceiveMemoryWarning {

    [super didReceiveMemoryWarning];
    // Releases the view if it doesn't have a superview
    // Release anything that's not essential, such as
cached data
}
```

These are `UIView` method stubs. We've seen them before. Again, the first one simply enables auto rotation. `didRecieveMemoryWarning` is simply the unmodified stub.

setTitle:, setDecimalValue: and setCurrencyValue:

```
- (void)setTitle:(NSString*)title {
    titleLabel.text = title;
}

- (void)setDecimalValue:(double)value {
    valueLabel.text = [NSString decimal:value];
}

- (void)setCurrencyValue:(double)value {
    valueLabel.text = [NSString currency:value];
}
```

These methods set our label's text. Notice that the value setters use the `NSString` category we defined earlier. Be sure to import `Formatter.h` at the top of this file.

dealloc

```
- (void)dealloc {

    [model release];
    [titleLabel release];
    [valueLabel release];

    [super dealloc];
}
```

Finally, we release our model and labels. That's it for these classes; however, we won't use them directly. Instead, we will make a subclass for each individual view. Go ahead and make a `MPGViewController`. I recommend basing it off the `NSObject` template. We won't need any of the `UIViewController` stubs. Edit `MPGViewController.h` as shown below.

MPGViewController.h

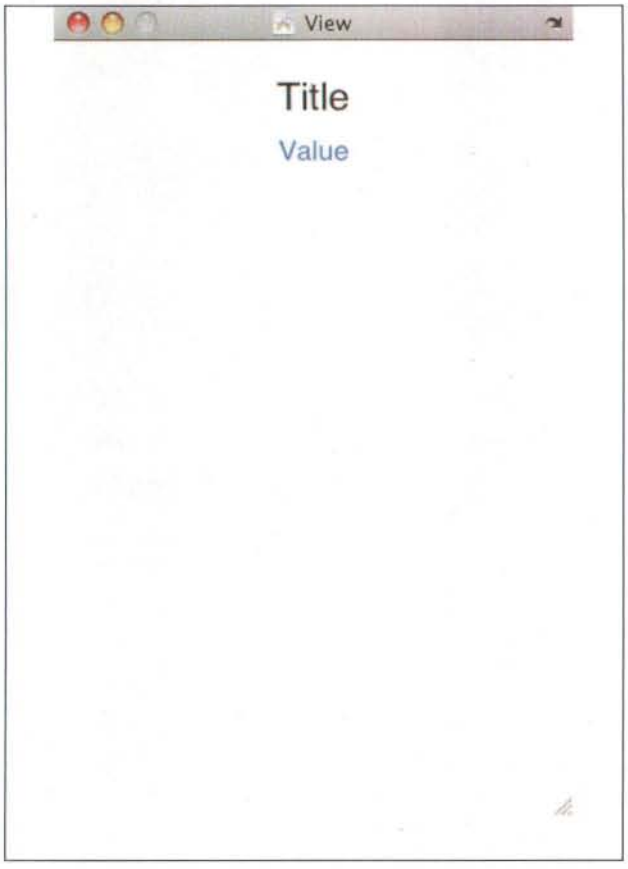
```
#import <UIKit/UIKit.h>
#import "StatsViewController.h"


@interface MPGViewController : StatsViewController {

}

@end
```

The implementation is almost as simple. We set the title once, when the view first loads, but we update the value each time the view appears.

MPGViewController.m

```
#import "MPGViewController.h"
#import "Model.h"

@implementation MPGViewController
- (void)viewDidLoad {

    [super viewDidLoad];
    [self setTitle:@"Miles Per Gallon"];
}

- (void)viewWillAppear:(BOOL)animated {

    [self setDecimalValue:[self.model milesPerGallon]];
}

@end
```

That's it for the code, now we just need to wire everything together. Save these files, then open `StatsView.xib`. Add two labels to the view, as shown below.



I recommend changing the Title's font size before positioning it. Open the fonts window by selecting Font fi Show Fonts. I changed the size to 24 pts. Once that's done, resize the label by selecting Layout fi Size To Fit. Then place the Label at the top of the view, and stretch it until it fills the view from margin to margin. You then center the text from the Attributes Inspector.

Note: the font controls are somewhat scattered. Font family, typeface and size are controlled by the Fonts window. Alignment and color are in the Attributes Inspector.

Now, change the File's Owner's class to `MPGViewController`. Make sure the view, `titleLabel` and `valueLabel` outlets are connected properly, and save the nib file. That's half our connections.

Open `MainWindow.xib`. Single click the MPG tab, and make sure the Inspector is showing the View Controller Identity information. Change the class to `MPGViewController`. Next, right click the tab, and make sure the model outlet is connected to the model object.

That's it. Save everything, then launch the application and take it for a spin.

Conclusion

Now, we've covered a lot of ground in a very short time. Let's quickly review the main points:

We managed our view hierarchy by pushing and popping views from the navigation controller.

We validated and formatted the text in our text fields.

We created a delegate to both fill our table and manage row selections.

We added rows to and removed rows from our table, including animations.

We wrote a category to extend an existing class.

We wrote an extension to add private methods to a class.

We examined the finer points of memory management for `IBOutlets`.

Of course, there's still a lot of work to be done. There's no way to edit an entry, and there's no way to set a custom date. The application also desperately needs more testing. For example, the current model works fine for a dozen or so entries, but what happens when the user enters hundreds or thousands?

Still, I wanted to show you something that was more than just a toy project. I hope working on GasTracker has given you a more-complete view of the entire iPhone app development process.

So, that's it. Get out there and make something great.

About The Author

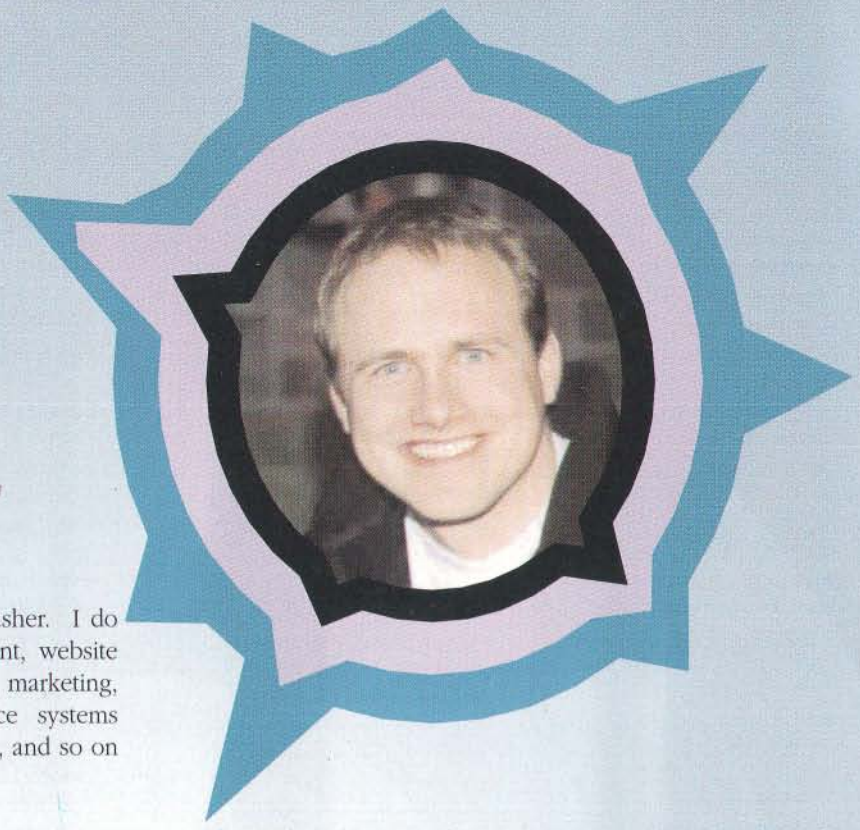
Rich Warren lives in Honolulu, Hawaii with his wife, Mika, daughter, Haruko, and his son, Kai. He is an associate scientist with 21st Century Systems, Inc., a freelance writer and a part time graduate student. When not playing on the beach with his kids, he is probably writing, coding or doing research on his MacBook Pro. You can reach Rich at rikiwarren@mac.com, check out his blog at http://freelancemadscience.blogspot.com/ or follow him at http://twitter.com/rikiwarren

Advertiser/Product Index

THE MACTECH SPOTLIGHT

Cortis Clark

Sol Robots L.L.C.



What do you do?

I'm the president or chief cook and bottle washer. I do pretty much everything here: product development, website development, product design and art, testing, sales, marketing, technical support, customer service, back office systems development, financial forecasting, mail room clerk, and so on and so forth.

How long have you been doing what you do?

I've been doing it since 2001. Part time at first, while the lead programmer at REAL Software, and then starting full time in January 2005.

Your first computer:

Apple IIe

Are you Mac-only, or a multi-platform person?

Most of my software runs on a PC as well, but I do all of my development and business operations on the Mac. In 2008, I've spent considerable time working on iPhone apps, and to a smaller degree, apps for Google's Android phone, all of the development for these again is on a Mac.

What attracts you to working on the Mac?

It's fun. I never had as much fun when I worked at jobs with PCs. There is a lot of energy and excitement in the Mac market. I also find, that despite the market advantage of Windows, it's far easier to make money in the Mac market. My Mac sales are easily 3x my Windows sales.

What's the coolest thing about the Mac?

I've a hard time limiting myself just to one "coolest" thing - so I've picked two: Exposé and Time Machine. I tend to have a lot of programs and windows open (I currently have 15 programs running) so Exposé really comes in handy to get to the right one. I find that Time Machine works great for development. When I'm heading down an unproductive path with my source code, I can use Time Machine to go back to an earlier version. Version Control is awesome, and can do even more, but it only works when you remember to use it. Time Machine works even if you don't.

What is the advice you'd give to someone trying to get into this line of work today?

Be persistent. You may not have a #1 bestseller the first time out. My first independent venture was available for six months before I had any purchases.

What's the coolest tech thing you've done using OS X?

The coolest tech I've done for OS X is the debugger I wrote for iota-calc. It allows you to see what is going on in a complicated arithmetic expression. I've never seen anything like it - before or since. It even lets you back-step.

Ever?

I built an image scanner out of Legos. I put a light/dark sensor on an arm. The arm would move back and forth to scan a line. The scanner was mounted on wheels which drove over the page to get the full picture. It had terrible resolution and was really slow, but it worked! It was loads of fun to make.

Where can we see a sample of your work?

http://solrobots.com
http://savebenjis.com

What is the next way you'll impact the Mac universe?

That's all hush-hush. :)

MT

If you or someone you know belongs in the MacTech Spotlight, let us know! Send details to editorial@mactech.com

get the latest now

» pay for it later *(without credit card debt)*

How? With a Business Lease from Small Dog Electronics.

Don't put off getting new equipment for your business. Small Dog Electronics will outfit your small, medium or large business with everything you need–and within your budget.
Benefits include:

» multiple leasing plans & companies
Two major leasing companies & numerous plans are available to ensure a perfect fit.

» lease-specific product specials
Take advantage of special deals only offered with a lease agreement.

» flexible buy-out plans with no commitment to buy
Turn over your equipment as the need arises to always have the latest & greatest.

» and more

Plus, **every shipment outside of Vermont is always tax-free,** and our award-winning customer service staff is ready to answer any question you may have. Contact us today–we're confident that we have a plan that will suit your needs.

There's more online! Visit **Smalldog.com/finance**
Contact us for a free quote: email **Rob@smalldog.com** or call **800-511-MACS x620.**

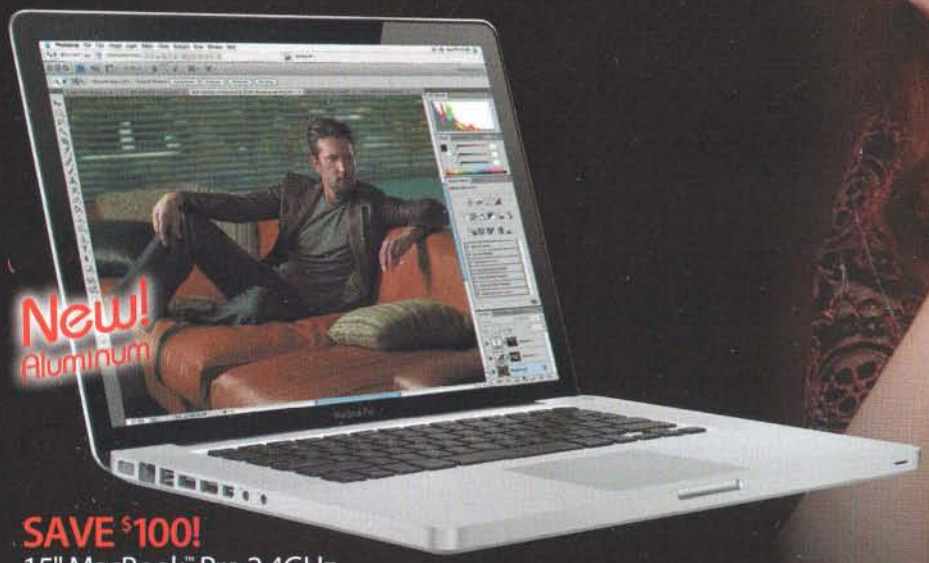
Small Dog Electronics
Always By Your Side

- **over 15 years in the Mac community**
- **3,000+ products for Macs + PCs**
- **5-star online merchant rating**
- **tax-free shopping outside of VT**

www.smalldog.com **800-511-MACS** Apple Specialist

Amp Up Your Stuff!™

Crank up your savings!
Max out your Mac & iPod!



SAVE $100!
15" MacBook™ Pro 2.4GHz
with 250GB Hard Drive and 2GB SDRAM
FREE Parallels Desktop!* FREE Epson Printer!*

~~$1994~~ - $100 mail-in rebate* = **$1894!*** #7684020
*After mail-in rebate. See our ad inside this magazine.

6 Months Same as Cash!
Offer valid for purchases over $500. Call for details.
Up to $200 Cash Back!*
On Apple computers. After mail-in rebate.
FREE Parallels Desktop!*
After mail-in rebate with purchase of an Apple computer.
FREE Epson Printer!*
After mail-in rebate with purchase of an Apple® computer.
*See our ad inside for details.

SAVE $594!
13" MacBook™ Air 1.6GHz
80GB Hard Drive
FREE Parallels Desktop!*
FREE Epson Printer!*
original price $1794
~~$1299.00~~ - $150 = **$1199.99!***
#7373085 *After mail-in rebate.

SAVE $644!
15" MacBook Pro 2.4GHz
9600M SuperDrive
FREE Parallels Desktop!*
FREE Epson Printer!*
original price $1994
~~$1499~~ - $150 = **$1349.99!***
#7691239 *After mail-in rebate.

SAVE $50!
13" MacBook™ 2GHz
SuperDrive™
FREE Parallels Desktop!*
FREE Epson Printer!*
~~$1294~~ - $50 = **$1244!***
#7684018 *After mail-in rebate.

SAVE $35!
Microsoft Office 2008
for the Mac Student
and Teacher Edition
list price ~~$149.95~~
now **$114.99!** #7352258

SAVE $76!
1TB LaCie Big Disk
Extreme+ USB 2.0,
FireWire 800 and
FireWire 400 Hard Drive
was ~~$259.95~~
now **$183.94!** #7274662

Authorized Reseller

See our ad inside this magazine.
Source code: MACTECH

MacMall®
Your #1 Apple Superstore!

Call 1-877-233-2838 or visit macmall.com